Int. J. Inf. Secur. (2017) 16:435–457
DOI 10.1007/s10207-016-0334-0

CrossMark

REGULAR CONTRIBUTION

Designing vulnerability testing tools for web services: approach, components, and tools

Nuno Antunes[1] · Marco Vieira[1]

Published online: 14 June 2016
© Springer-Verlag Berlin Heidelberg 2016

Abstract This paper proposes a generic approach for designing vulnerability testing tools for web services, which includes the definition of the testing procedure and the tool components. Based on the proposed approach, we present the design of three innovative testing tools that implement three complementary techniques (improved penetration testing, attack signatures and interface monitoring, and runtime anomaly detection) for detecting injection vulnerabilities, thus offering an extensive support for different scenarios. A case study has been designed to demonstrate the tools for the particular case of SQL Injection vulnerabilities. The experimental evaluation demonstrates that the tools can effectively be used in different scenarios and that they outperform well-known commercial tools by achieving higher detection coverage and lower false-positive rates.

Keywords Software vulnerabilities · Vulnerability detection · Security testing · Web services

1 Introduction

Web services (WS) are nowadays used to support the information systems of wide range of organizations in sectors such as banking and manufacturing, representing a strategic mean for data exchange, content distribution, and systems integration [1]. They are supported by a complex software infrastructure, which typically includes an application server, the operating system, and a set of external systems (e.g. other services, databases, and payment gateways). Web services are one of the cornerstones of service-oriented architecture (SOA), making them the *lingua franca* for systems integration.

The security of web applications is, in general, quite poor [2,3]. Web services are no exception, and research and practice show that web services are often deployed with software bugs (i.e. vulnerabilities) that can be maliciously exploited [4]. Injection vulnerabilities, consisting of improper code that allows the attacker to inject and execute commands, enabling, for instance, access to critical data, are particularly frequent [2]. SQL Injection and XPath Injection vulnerabilities are especially relevant, as services frequently use a data persistence solution based in a database [5] or in a XML solution [6].

Vulnerability testing tools provide an automatic mean to search for vulnerabilities while avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand. However, even state-of-the-art tools frequently present low effectiveness in terms of both vulnerability **detection coverage** (ratio between the number of vulnerabilities detected and the total number of existing vulnerabilities) and **false-positive rate** (ratio between the number of vulnerabilities reported that do not exist and the total number of vulnerabilities reported) [4,7].

The **main problem** is that most vulnerability testing tools try to be as generic as possible (to detect many types of vulnerabilities), but are typically very limited in terms of the detection approaches they implement for each vulnerability type and do not take advantage of the specific access conditions to the target services. In fact, due to cost and time restrictions, generic tools are often selected as they can detect a wide spectrum of vulnerabilities, although with reduced

✉ Nuno Antunes
nmsa@dei.uc.pt

Marco Vieira
mvieira@dei.uc.pt

[1] Department of Informatics Engineering, University of Coimbra, Polo II - Pinhal de Marrocos, 3030-290 Coimbra, Portugal

effectiveness. Additionally, there are specific characteristics that distinguish web services from other web applications and that influence the development of vulnerability detection tools, namely the well-defined interface, the need for testing third-party web services, and the interoperability and reduced dependency among services.

Building effective tools requires innovative techniques that take into account the different access conditions to the services under testing. For example, if one has access to the web service interfaces, including interfaces with external resources like other services or databases, then an improved technique based on interface monitoring may be used in detriment of traditional penetration testing. Furthermore, testing tools should implement a generic procedure that supports the design of standardized and modular tools, guaranteeing an easier way to develop new tools and to improve existing ones, *allowing more effective detection of vulnerabilities*. The idea is that the different components should implement specific features of the tool, in a decoupled and modular manner, allowing for easily designing and later improving the tool. It also allows web service developers to modify existing tools to better fit their needs, just by modifying some particular components.

In this paper we **propose a generic approach for designing vulnerability testing tools for web services.** The approach *is based on modular development* and defines the components and the testing procedure that a tool should implement. The components include a workload emulator (responsible for generating and executing a set of requests to exercise the web service), an attack emulator (in charge of generating and injecting requests that simulate attacks), a service monitor (in charge of instrumenting the service under testing, if needed, and collecting relevant information to support vulnerabilities identification), and a vulnerability detector (responsible for analysing the collected information and identify vulnerabilities, and for running the testing procedure). Based on the proposed approach, we design three specific testing tools that implement three complementary techniques for detecting injection vulnerabilities in web services, thus providing an extensive support for developers to test different scenarios.

The evaluation of the approach includes an analysis of the modularity of the implemented tools and a case study designed to demonstrate that the implemented tools are able to perform as well as the tools implemented in *ad hoc* fashion. The modularity analysis ought to show the **high cohesion** of the defined modules and the **loose coupling** between them. The case study uses a reference set of 80 operation from 21 services (adopted from the benchmark for vulnerability detection tools presented in [8]). Although the case study is focused in *SQL Injection*, most of the concepts and techniques presented can be easily adapted for other kinds of *injection vulnerabilities* (the most frequent ones in web applications and web services [4]). Results clearly show that the three detection techniques implemented have different performances depending on the web service access conditions. Those results were compared with the ones obtained by three commercial vulnerability scanners, showing that our tools perform consistently better than those scanners. Besides showing the effectiveness of the tools, the case study presented demonstrates the applicability of the approach for designing vulnerability testing tools for web services, as it was able to accommodate the design of three different tools.

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

– The proposal of a generic approach for designing vulnerability testing tools for web services, which includes the definition of the testing procedure and of the tool components. By instantiating this approach, it is possible to easily design different tools targeting different types of vulnerabilities, reuse components from other tools to develop new tools faster, and evolve existing tools by proposing improved versions of different components;
– An extensive survey of the related work on vulnerability detection tools and techniques targeting web services and web applications, both on commercial products and on proposals from research initiatives;
– The redesign of three vulnerability testing techniques previous published [9–11] to instantiate the proposed generic approach and components. Although all these techniques target the detection of injection vulnerabilities in web services, they are based in very different vulnerability detection techniques and allow us to demonstrate the flexibility of the approach.
– The experimental evaluation of the tools based on a case study allows to evaluate the redesigned tools and compare their effectiveness with state-of-the-art penetration testing tools for vulnerability detection on web services.

The outline of the rest of this paper is as follows. Section 2 introduces relevant background and related work. Section 3 describes the overall approach for designing vulnerability scanning tools for web services and defines the required components. Section 4 discusses the design of the three tools for detecting injection vulnerabilities. Section 5 presents the evaluation of the approach and of the implemented tools. Section 6 discusses the shortcomings of the paper and ideas for improvements. Finally, Sect. 7 concludes the paper and puts forward future work.

2 Background and related work

Web services provide a simple interface between a provider and a consumer [12] and are supported by open protocols such as the Simple Object Access Protocol (SOAP) [12],

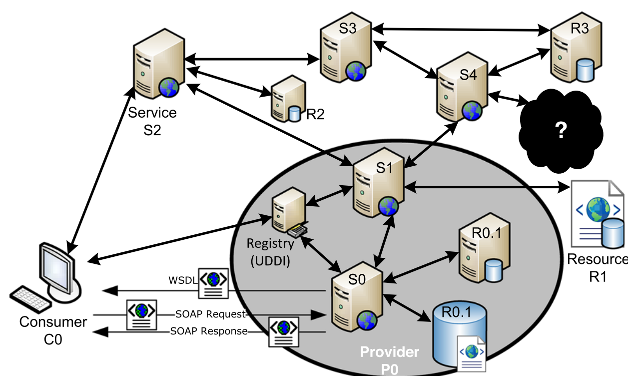

Fig. 1 Web services typical structure

which are used for exchanging XML-based messages between the consumer and the provider over the network (using for example http or https protocols). A web service may include several operations that are defined in a description file (e.g. using WSDL [13] or WADL [14]), which is used to generate server and client code, and for configuration. A broker is used to enable applications to find web services. Figure 1 depicts a typical web services environment.

2.1 Web services threats and challenges

Published studies show that, in general, web applications present dangerous security flaws. The ten most critical web application security vulnerabilities [15] were presented in 2010 by the Open Web Application Security Project (OWASP Foundation) [16]. At the top of this list, we can find vulnerabilities that are also reported as the two most found in other studies [17,18], consisting of injection and cross-site scripting (XSS). XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. Although some complex applications use data from web services in session aware contexts, most times web services work without any attachment to websites. Thus, *XSS is not a priority in services environments* [4].

On the other hand, *injection vulnerabilities are critical*. An injection occurs when data from an input are used without any check as part of a command or of a query. Issues like SQL Injection and XPath Injection are particularly relevant in web services as they are directly related to the way the web service code is structured [15,19]. Basically, SQL Injection and XPath Injection attacks take advantage of improper coded applications to change SQL commands that are sent to the database or tamper XPath queries used to access parts of an XML document. Several studies [4,20,21] show that a large number of web services are deployed with security flaws that range from code vulnerabilities (e.g. code injection vulnerabilities) to the incorrect use of security standards and protocols. To minimize security issues, developers should use vulnerability testing tools to search applications for vulnerabilities.

Several techniques for the detection of software vulnerabilities are available for web applications [19]. Although most of these approaches can also be applied to web services, there are several specificities that distinguish web services from other web applications [22] and that should be kept in mind when developing and using vulnerability detection tools: (1) web services have always a **well-defined interface**. This is mostly a positive aspect, as it avoids the need for a crawling phase (required by some approaches to learn the interface of a web application), but makes it easier to mask information about internal problems of the application, which can be a limiting factor, as it reduces the information when compared to what testing tool can extract from the service's responses during crawling. (2) In many situations, the user that needs to test the security of a web service is not the owner and thus **cannot access its internals**, which is a requisite for some vulnerability detection techniques. A common example of this scenario is when a consumer has to select a service from a multitude of alternatives provided by third parties. (3) The **interoperability and reduced dependency** among services not only facilitates their replacement or modification without requiring changes in other parts of the system, but also requires vulnerability detection to be effective, i.e. allowing an easy way to test and retest the services of the system while taking into account a more complete view of system and the potential interactions among services.

2.2 Black-box testing

Most vulnerability testing tools are based on black-box testing [19,23], which consists of the analysis of the program execution from an external point of view. In short, it consists of exercising the software and comparing the execution outcome with the expected result and is one of the most used techniques [24,25] for *verification and validation* of software. **Penetration testing**, a specialization of testing, is based on the analysis of the program outputs in the presence of malicious inputs, searching for potential vulnerabilities. In this approach, the tester does not know the internals of the web application and it uses fuzzing techniques over the HTTP requests [19]. The tester needs no knowledge about the implementation details and tests the inputs of the application from the user's point of view. The number of tests can reach hundreds or even thousands.

The most common automated penetration testing tools used in web applications are generally referred to as **web security scanners** (or web vulnerability scanners). These scanners have a predefined set of test cases that are adapted to the application to be tested, saving the user from defining all the tests to be done. In practice, the user only needs to

configure the scanner and let it test the application. Once the test is completed, the scanner reports existing vulnerabilities (if any are detected). Most vulnerability scanners are commercial tools, but there are also some free applications. In practice, three brands lead the market: *HP WebInspect* [26], *IBM Rational AppScan* [27], and *Acunetix Web Vulnerability Scanner* [28]. On the other side, popular free security scanners that support web services are *Foundstone WSDigger* [29] and *WSFuzzer* [30]. The main problem of these open-source tools is that, in fact, they do not detect vulnerabilities: they only automate the tests. In other words, they submit attacks to the web service and log the responses leaving the task of examining these logs.

Regarding state-of-the-art research proposals, multiple techniques have been presented, as follows. *WAVES* [31] is a black-box technique for testing web applications for SQL Injection vulnerabilities. The technique is based on a reverse engineering process that identifies the data entry points of the application and attacks them using malicious patterns. An algorithm is proposed to allow "deep injection" and to eliminate false negatives. During the attack phase, the responses of the application to the attacks are monitored and machine learning techniques are used to improve the attack methodology.

SecuBat [32] is an open-source web vulnerability scanner that uses a black-box approach to crawl and scan websites for the presence of exploitable SQL Injection and XSS vulnerabilities. SecuBat does not rely on a database of known bugs. Instead, it tries to exploit distinctive application-level vulnerabilities by issuing attacks targeting these vulnerabilities, including proof-of-concept exploits in certain cases. After the attack is launched, the response is parsed and interpreted in an attempt to find attack-specific response criteria and keywords. Based on this process, it is calculated a confidence value on the success of the attack. False positives are possible, thus requiring a careful tuning of the confidence value threshold.

The problem is that both *WAVES* [31] and *SecuBat* [32] can only be applied to web applications (not to web services, where the interface is well defined) and ignore user knowledge about the application being tested.

An evaluation of web vulnerability scanners is presented in [33]. Both commercial and open-source scanners were evaluated, in a total of 11 scanners. To test the tools, the authors introduced different types of vulnerabilities in a realistic web application, challenging the crawling capabilities of the tools. The main findings of the study were that (1) the crawling process is critical to the success of the scanning process and that (2) many classes of vulnerabilities are completely overlooked by these tools. Due to the characteristics of web services, the difficulties related to the crawling process are not an issue, as the interface is well defined. However, a limitation related to this one is the code coverage of the tests. This means that if the tests do not exercise correctly the web service code, they are not able to exploit the vulnerability and also not able to detect it. A final limitation of black-box approaches in web services environment is that the vulnerability detection is limited by the output information of the application. It is important to keep in mind that the output information includes not only to the web service responses, but also other information available to the exterior (e.g. response time). This work contributes to an effort on overcoming some of these limitations: on the one hand, the tools presented are based on techniques with improved visibility on the internals of the web services, and on the other hand, the modularity of the designs proposed facilitate the process of improving the test coverage by replacing the workload and attack load generation modules by better ones.

2.3 Grey-box testing

Grey-box vulnerability detection approaches try to overcome some of these limitations by monitoring the behaviour of the application during the execution of the tests, trying to find anomalies caused by vulnerabilities present in the code. The idea is that by analysing the internal behaviour of the code in the presence of realistic inputs it is possible to identify bugs and vulnerabilities. The effectiveness of such analysis depends strongly on the input values (as in testing), but it takes advantage of the observation of the source code. For improving the effectiveness of the analysis, the program must be executed with sufficient test inputs. Code coverage analysers help guaranteeing an adequate coverage of the source code [34,35].

While other works focused on identifying vulnerabilities related to the use of external inputs without sanitizations, the work presented in [36] introduces an approach that combines static and dynamic analysis techniques to analyse the correctness of sanitization processes in web applications. First, a technique based on static analysis models the modifications that the inputs suffer along the code paths. This approach uses a conservative model of string operations, which might lead to false positives. Then, a technique based on dynamic analysis works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process.

Other techniques to detect SQL Injection attacks need the application developers to change the code in order to insert the detection mechanisms. SQLCheck, presented in [37], checks SQL queries at runtime to see if they conform to a model of expected SQL queries. The model is expressed as a context-free grammar that only accepts legal queries. The user supplied portions are marked in queries with a secret key. A parser is generated based on the grammar. At runtime, the parser verifies the generated query and when a SQL Injec-

tion attack is detected, the request is rejected. The security depends on the generated keys, and the approach requires the developer to rewrite code to manually insert the secret keys into dynamically generated SQL queries. Another problem of this technique is that it introduces additional complexity to the development phase, which limits its applicability.

Runtime anomaly detection tools can be used for both attack detection and vulnerability detection. One of those tools is AMNESIA (Analysis and Monitoring for NEutralizing SQL Injection Attacks) [38] that combines static analysis and runtime monitoring to detect and avoid SQL Injection attacks. Static analysis is used to analyse the source code of a given web application building a model of the legitimate queries that such application can generate. At runtime, AMNESIA monitors all dynamically generated queries and checks them for compliance with the statically generated model. When a query that violates the model is detected, it is classified as an attack and is prevented from accessing the database. The problem is that the model built during the static code analysis may be incomplete because it lacks a dynamic view of the runtime behaviour.

3 Overall approach and components

Although most of the works discussed above propose new and more effective techniques for vulnerability and attack detection, none of them follows a standard for designing the tools, following instead specific architectures. This reduces their *maintainability*, making it harder to introduce future improvements. It also limits the *reuse of their components* which does not allow taking advantage of them when proposing new tools that could benefit from a faster development and from the knowledge “embedded” in the tools. Again, it is important to emphasize that the goal of this paper is not to propose new techniques: instead, the goal is to present a generic modular approach for designing tools and describe the steps necessary to develop tools following this approach. In this sense and in terms of vulnerability testing, the novelty of the paper is more on the architectural design of vulnerability detection tools than on new and more effective techniques.

Designing an effective vulnerability testing tool requires the clear definition of a set of components and of a *standardized* testing procedure. Although other techniques and tools follow designs that share similarities to our proposal, they were designed in an *ad hoc* fashion without any formalism or standard procedure [26–30]. The reasoning behind this proposal is that the tools must be composed of different components that should implement specific features of the tool, in a decoupled manner. Although this may be common sense, the fact that most tools do not follow any kind of standardized approach demonstrates the value of the proposal. Adopting this kind of design strategy brings many advantages when compared with the isolated development of vulnerabilities detection tools, as summarized in the following:

- It facilitates the design and maintenance of the tools, including iterative improvements. In fact, to improve the effectiveness of the tool, it is only necessary to upgrade each module by using newly developed techniques.
- It makes easier and faster the task of developing new tools. With an easier design, it comes an easier development process and developers can also benefit from reusing state-of-the-art modules in which their techniques will not introduce innovations.
- It also allows developing teams to modify the vulnerability detection tools to better fit their needs, just by modifying some particular components.
- It makes possible the combination of multiple vulnerability detection tools, as they share the same design principles, applying them to detect different types of vulnerabilities. The advantage is that this allows detecting a wider spectrum of vulnerabilities and vulnerability types while maintaining the higher effectiveness of specialized tools.

Figure 2 shows the relation among the proposed components and with the web service under testing. As we can see, the workload emulator and the attack emulator work together to create and submit attacks, the vulnerability detector uses knowledge about the attacks and information collected from the web service to identify vulnerabilities, and the service monitor is in charge of instrumenting the web service and collecting information to feed the vulnerability detector. The vulnerability detector is also in charge of implementing the testing procedure by coordinating the remaining components.

Due to the high diversity of web services, types of vulnerabilities, and vulnerability detection approaches, in this paper we argue that designing an effective tool requires focusing on a well-defined domain. In fact, the division of the spectrum into well-defined areas is necessary to allow making choices during the definition of the components and procedure. In the context of this work, the **definition of the tool domain** includes selecting the type of web services, the type of vulnerabilities to detect, and the vulnerability detection approach. The first, class of web services (e.g. SOAP, REST), allows understanding the characteristics of the services that will be tested. The second defines the types of vulnerabilities (e.g. SQL Injection, XPath Injection, file execution) that should be detected by the tool. Finally, the vulnerability detection approach (e.g. penetration testing, anomaly detection) specifies the approaches that the tool will use to detect vulnerabilities.

The next subsections detail the proposed components (always referring to Fig. 2). The testing procedure is dis-

Legend:
WE - Workload Emulator
- User input
WG - Workload generator
WI - Workload injector
WG Ext. - External workload generator
AE - Attack Emulator
AG - Attack generator
AI - Attack injector
AG Ext. - External attack generator
VD - Vulnerability detection module
TD - Testing driver
VI - Vulnerability identifier
Report - Injection vulnerabilities detected
SM - Service monitor
IC - Information collector
SI - Service instrumentation
IM - Information manager
SP - Service Provider
WS - Web service under test
R0 - XML database
R1 - Payment gateway
R2 - Database
R3 - FTP server
- - - Optional elements or connections

Fig. 2 Generic architecture of a web service vulnerability testing tool

cussed in Sect. 3.4, together with the vulnerability detector component. Note that the goal of this section is not to design a specific tool (that is done in Sect. 4), but to propose an approach that can be generically used to design testing tools for detecting vulnerabilities in web services in general, and thus, specific details are left out and discussed later.

3.1 Workload emulator

The **workload emulator** (WE) component is in charge of generating a set of valid requests. These requests will be used by the attack injector for generating attacks, but can also be executed in the absence of attacks to exercise each operation of the web service under testing. The motivation for the decoupling of workload and attack emulators is many fold. First, the non-malicious requests (the workload) are very useful to *study the web service regarding its correct functioning*. This knowledge is of utmost importance for some vulnerability detection processes as it allows them to detect anomalies or discard behaviours that seem related to a vulnerability but are also observable when the service is not under attack (e.g. a database error that happens in situations in which the service is not under a SQL Injection attack). In second place, it allows an easy *independent improvement of the modules with all the benefits for the process*. In fact, the generation of good workloads is not dependent on security knowledge, meaning that it is possible to have researchers or developers with experience in functional tests improving the workload generation modules and this will also have very positive effects on the attack generation, improving the overall technique. Finally, the decoupling *gives more flexibility to the developers*, allowing them to adapt the design to their technique and integrate third-party modules or tools when necessary.

The WE includes two elements: a workload generator and a workload injector.

The **workload generator** (WG) starts by obtaining the required definitions about the web service under testing. As mentioned before, a web service interface is described in a descriptor file (e.g. WSDL, WADL) [12], which should be processed to obtain the list of operations, parameters (including return values), and associated data types. However, as in most cases the valid values for each parameter (i.e. the domain restrictions of the parameter) are not available in that file and associated schemas, the user should be allowed to provide additional information about the valid domains for each parameter (including for parameters based on complex data types, which are composed by a set of individual parameters). Note that, for web service operations with several input parameters, the valid domain for a given parameter may be dependent on the value specified for another parameter (this must also be specified). This information would also benefit tools that do not follow this design (e.g. commercial scanners), but the modularity of this approach allows easily modifying or upgrading the WG module to deal with the existing information or even to ask the user if he is interested in providing it.

A workload (set of valid web service calls) should then be generated to exercise each operation of the web service under testing. As it is not possible to define a generic workload that fits all web services, we need to apply a specific workload to each specific service. Three alternatives are available for implementing the workload generation (a vulnerability testing tool may support more than one alternative, thus offering to the user the option of selecting the one that best fits his requirements). The first option is to use a *user-defined workload generator*. In this case, the user of the tool should implement a generation component based on the knowledge he has about the service being tested. The workload emulator should provide an easy way for integrating this generation component, which only needs to interact with the workload injector (in charge of submitting the workload requests to the services under testing) and with the attack generator (that creates attacks based on the workload requests in an educated manner, as described in the next subsection).

The second option is to *use the functions of an existing vulnerability scanner*, which consists of supporting the integration of external tools in a similar way to the user-defined

workload. A key aspect is that in this case we are not interested in the attack generation and vulnerability detection capabilities of such tool (this will be addressed later in the design of the attack injector and of the vulnerability detector), but only on interface identification and workload generation features. Additionally, note that selecting an existing vulnerability scanner to provide this feature may not be an easy task as, depending on the type of vulnerabilities to detect and on the detection approach to implement, existing scanners may be limited in the support provided also because they were not developed in a modular standardized fashion that would allow them to be integrated with other tools.

The last option is to include in the vulnerability testing tool a *module that implements a workload generator*. Several approaches can be used for generating web service requests, including (see [39] for details on these approaches) deterministic generation (e.g. based on constant values, step functions, values shuffling), stochastic methods (e.g. based on uniformly distributed random values, random values added by a step function, and Gaussian, Poisson, or exponential distributions), and hybrid approaches (a combination of multiple approaches). As the goal is to generate (valid) requests that adequately exercise the services under testing (i.e. allow achieving a high coverage of the code under testing), this process should take into account the web service definitions mentioned above. It is also of extreme importance to design a workload generator that satisfies the requirements in terms of the target vulnerabilities and of the detection approach being implemented.

The representativeness of the workloads used is crucial in the effectiveness of the vulnerability testing process. If the inputs used are not able to exercise the code completely, the technique may also not be able to trigger and detect the vulnerabilities. This way, the developers of the vulnerability testing tool should select the workload generation strategy that best fit their scenarios. A key advantage of the modularity of a tool is that it allow selecting the most favourable options for the workload generator or even develop a new and potentially more adequate one.

The **workload injector** (WI) component takes the workload generated and submits it to the web service. This is an optional element, as some approaches may not require the execution of a workload. For example, classical penetration testing is based only on the execution of the penetration tests. On the other hand, anomaly detection approaches require a profiling phase; thus, a workload execution is required. A feature that may be added to the workload injector is code coverage analysis [34,35]. The idea is that in the cases where source code or bytecode is available, code coverage can be used to drive the generation of the workload requests. This can be done by using a tool that analyses the execution profile of the web service (during the execution of the workload) and calculates a coverage value (many code coverage metrics can be used in this context; see [40] for an overview), which can then be used to decide whether more requests are needed to increase coverage. In such case, the injector component should ask the generator component to create additional requests.

3.2 Attack emulator

The **attack emulator** (AE) component is in charge of automatically generating attacks and of submitting them to the web services under testing. This way, it includes two elements: an attack generator, in charge of creating attacks, and an attack injector, responsible for submitting those attacks.

Two options can be considered for implementing the **attack generator** (AG). The first is to *use the functions of an external vulnerability scanner*, which consists of supporting the integration of external generators in a similar way to what is done for the workload generation (see previous section). Note that, similarly to the workload generator, selecting an adequate external attack generator may be difficult, as existing tools may not implement the testing strategies required to successfully implementing a given vulnerability detection approach and because these tools can be hard to integrate, as they were not developed following a modular standardized style.

The second option is to *include an internal attack generator* that takes the workload and replaces valid values by malicious values following an extensive set of mutation rules (see Table 1 for examples of typical mutation rules). By building the attack load on top of the workload, it is possible to avoid functional redundancy (i.e. repeated implementation of code for request generation that is common to both workload and attack load generation) and take advantage of the knowledge on test generation that it as embedded on it inherently. Also, it is possible to take advantage of future improvements of the workload generator module without changing the attack emulator module. Obviously, the mutation rules depend on the type of vulnerabilities to detect (the table shows different examples for SQL Injection and XPath Injection) and should be as complete as possible in order to achieve high detection coverage. This way, defining the set of mutation rules is a complex task that should take into account multiple sources of information, including information on how existing tools work, knowledge on previous successful attack attempts in the field, and scientific references.

Although the process for generating the attacks may depend on the vulnerability detection technique, we propose a generic procedure whose goal is to support the design of generation approaches capable of creating comprehensive sets of attacks. As shown in Fig. 3, such procedure includes several phases, where each phase focuses on generating malicious calls that target a given operation and includes a set of steps. Each step targets a specific parameter of the opera-

Table 1 Examples of SQL Injection attack types

	SQL Injection attack
(1)	`" or 1=1 --`
(2)	`" or 1=1 or","="`
(3)	`' or (EXISTS)`
(4)	`' or uname like'%`
(5)	`' or userid like'%`
(6)	`' or username like'%`
(7)	`' UNION ALL SELECT`
(8)	`' UNION SELECT`
(9)	`char%2839%29%2b%28SELECT`
(10)	`" or 1=1 or ""="`
(11)	`' or ''='`

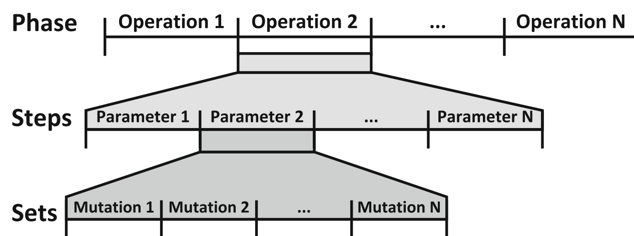

Fig. 3 Generic process for generating the attacks

tion and comprises several attack sets. An attack set includes the attacks to be performed over a given parameter, which are generated by applying the mutation rules. Obviously, the same mutation rule may be applied one or more times over the same input parameter.

The **attack injector** (AI) component is in charge of submitting the generated attacks to the web service under testing. If an external attack generator is used, then the injector should provide the required integration interfaces. Similarly to the workload injector, the attack injector may also support coverage analysis features, whose output can be used to drive the generation of additional attacks.

3.3 Service monitor

To identify vulnerabilities, we need to collect as much information as possible about the behaviour of the web services under testing. This information is later used by the vulnerability detector component and the type of information that can be collected depends on the access conditions to the target web services. In fact, as these environments are typically based on services that can be under the control of multiple providers, the users of the testing tool may have different levels of access to the services to be tested. In practice, three scenarios may be considered (see example in Fig. 1):

- *Scenario 1* a service is within reach but not under control, like in the case of *S2* in Fig. 1. Only the external interface of the service is known, and the input information is available. This means that it is not possible to access the source code and only black-box testing techniques can be used. This scenario also simulates the point of view of the consumer.
- *Scenario 2* a service is under control but some of the resources that it uses are not, like in the case of *S1* (it uses resource R1 that is not under control). In this case, it may not be possible to access the source code. However, all the interfaces between the service and the external environment are known. This allows obtaining extra information about the domains of web service's inputs.
- *Scenario 3* a service is under control and also the resources that it uses are, like in the case of *S0*. It is possible to use all kinds of vulnerability detection techniques, including the ones that require access to the source code.

Each vulnerability detection technique has its specific requirements in terms of the information needed, but the most basic information is the web service requests and corresponding responses (to both workload requests and attacks). This request/response can be used together with other interactions to disclose more advanced vulnerabilities, a classic example is to perform a *tautology* attack [41] that can be more easily detected using one request that will be evaluated to true and another that will be evaluated to false, and then compare the results. In the case of more advanced approaches, additional information may be related to the internal functioning of the web service, to the web services interfaces (including interfaces with external resources like other services and databases), etc. This way, the service monitor should be able to instrument the target services in a way that allows collecting the required information, in the less intrusive way possible. Depending on the type of information needed and on the level of access to the internals of the web service, there are multiple options to monitor web services. Three examples are: perform network packet sniffing (consists of reading each packet as it flows across the network), use a proxy (serving as a relay for HTTP requests), and apply code instrumentation (instrument the service code to include monitoring facilities).

In practice, the **service monitor** (SM) component should consider three components: **service instrumentation** (SI), an optional component in charge of instrumenting the service under testing, as needed; **information collector** (IC), responsible for collecting the information during the execution of the tests; and **information manager** (IM), responsible for storing the collected information in a database and for providing that information to the vulnerability detector component when required.

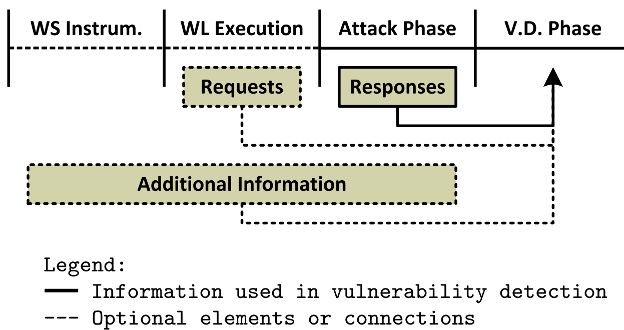

Fig. 4 Proposed testing procedure

3.4 Vulnerability detector

The **vulnerability detector** (VD) is in charge of processing and correlating the information collected to detect vulnerabilities. This component is probably the most critical one and should be able to identify as much vulnerabilities as possible (based on the available information), while minimizing the number of false positives reported. As mentioned before, in addition to the traditional analysis of requests and responses, vulnerability detection can be based on more elaborated approaches such as interface monitoring [10] and anomaly detection [11] (what is important is to apply techniques that adequately take advantage of the available information). The vulnerability detector is also the component responsible for managing all the tests by implementing the testing procedure. This is achieved by orchestrating the components presented before. This way, two elements should be included in the detector: the **vulnerability identifier** (VI) and the **testing driver** (TD). We propose to keep these two elements inside the same component as the testing procedure and the vulnerability detection approach greatly influence each other and are highly dependent on the vulnerability detection technique being implemented.

A key aspect is that the testing procedure should be as standard as possible in order to accommodate the vulnerability detection techniques of multiple different tools that, by their turn, will potentially support a more exhaustive testing and high vulnerability detection coverage of a broader range of vulnerabilities. Although such procedure depends on the specificities of the detection technique, in Fig. 4 we present an overview of the generic approach we propose. As shown, the procedure includes four phases: (1) web service instrumentation phase, (2) workload execution phase, (3) attack phase, and (4) vulnerability detection phase.

In the **web service instrumentation phase**, the service monitor component is asked to instrument the web service under testing in a way that allows gathering the required information. As service instrumentation may not be required in some techniques, this phase is optional. For example, in the case of classical penetration testing, the only information needed is web service requests and responses, whose collection does not require any particular instrumentation (this information is automatically provided by the workload injector and by the attack injector).

The **workload execution phase** consists of generating and submitting the workload requests. This phase is also optional as running a workload may not be needed in some cases. For example, classical penetration testing does not require the execution of the workload, but only the injection of the attacks. In practice, it consists of using the workload generator component for generating requests and the workload injector component for submitting those requests. The goal is to allow the service monitor component to gather information on the behaviour of the web service in the absence of attacks.

During the **attack phase**, attacks are generated and submitted to the web service. In practice, it involves using the attack generator component for generating attacks and the attacks injector component for submitting them. During this process, the service monitor gathers information about the behaviour of the web service in the presence of the attacks. This information, combined with the one collected during the workload execution phase, is then used during the **vulnerability detection phase** for identifying vulnerabilities by applying the defined detection technique. For example, the *SQL commands* that are issued to a database can be used by the vulnerability detector to search attack signatures that are an indication of an SQL Injection vulnerability, as will be detailed in Sect. 4.2.3.

4 Testing tools for injection vulnerabilities

In this section, we present the design of three vulnerability testing tools that implement the generic approach and components presented in the previous section. The application domain of these tools is SOAP web services and injection vulnerabilities. As mentioned before, this definition is of utmost importance and allows making the right decisions regarding the design of the components and procedure. The vulnerability detection techniques implemented by the tools presented in this section were previously introduced in [9], [10], and [11], respectively. In this work, we redesigned those techniques based on the approach presented in Sect.3, making them consistent in terms of the components and of the testing procedure.

Like most vulnerability testing tools, these three techniques were designed in an independent manner in their original prototypes, following an *ad hoc* design approach. This way, during this process it was necessary to analyse the existing tools, identify the modules of the generic architecture that made sense in each context, and proceed to redesign. After this step, it was necessary to re-implement the function-

alities of the tools following the new *modular* design. One of the key challenges was to decouple vulnerability identification logic from the other parts of the tools, as it was in all cases integrated with the testing code. After decoupling, it is necessary to feed the new module with the information necessary for its execution. Although this was a time- consuming task, the idea is that after that process, it is much easier to replace the VI module by an improved one that will be fed also with the information available.

In this context, *the main benefits of the redesign of the tools are flexibility and maintainability*. The introduction of improvements in previous versions was a time-consuming operation that required modifications across all the application. Instead, in the new versions it is only necessary to perform changes in the modules related. Additionally, it is very easy also to replace each module, something that was a very hard task due to the interdependence of the code blocks. To improve the representativeness of the tests applied (both the workload and the attacks) in the new versions of the tools, we only need to modify or replace the WG module. In the previous version, it would be necessary to do changes in all the workload-related code and many times also in the attack-related and vulnerability detection code. Another example is how easy it is to replace other modules, for example the vulnerability identifier (VI) module. For this, it is only necessary to replace the old module by a new one, as it will be ready to use the information provided by the testing driver (TD) and the information manager (IM).

The *corollary* or the *ultimate benefit* of this redesign process is that it will allow, in a near future, to integrate multiple approaches and techniques in just one integrated tool. By reusing the modules that are commons and selecting the other modules through simple configurations, it is possible to provide the user with the versatility to use multiple vulnerability testing approaches in different scenarios with the help of just one tool.

The following sections present the new architecture and modules of the redesigned versions of the tools.

4.1 Improved penetration testing (`IPT-WS`)

IPT-WS targets the detection of injection vulnerabilities in services within reach but not under control (i.e. Scenario 1 defined in Sect. 3.3). Comparing to existing web vulnerability scanners based on penetration testing, our approach has three key improvements: (1) we use a representative workload to exercise the services and understand the expected behaviour (i.e. the typical responses in the presence of valid inputs); (2) the set of attacks performed is a compilation of all the attacks performed by a large set of scanners plus many attack methods that can be found in the literature; and (3) we apply well-defined rules to analyse the web services responses in order to improve coverage and remove false positives. These rules include comparing the responses obtained when using malicious inputs with the normal responses (i.e. responses in the presence of a valid workload) and with the responses from robustness testing [42]. Figure 5 presents the tool design, which is detailed in the following subsections. The monitoring component is not included as the vulnerability detection depends only on the use of the web services requests and responses, which are directly provided by the workload emulator and by the attacks emulator.

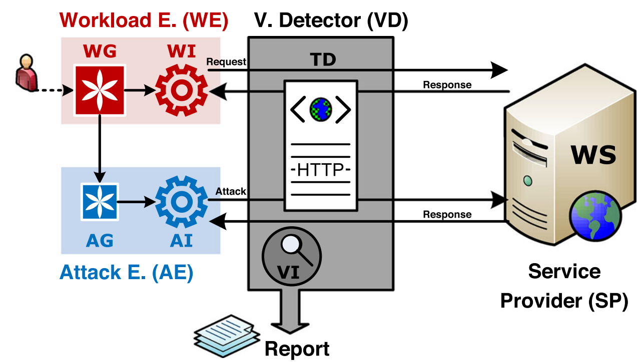

Fig. 5 IPT-WS overall design. (see Fig. 2 for module's names)

4.1.1 Workload emulation

For generating the workload, the tool automatically reads the web service definitions (i.e. operations, return values, parameters, data types, and domains) from the WSDL file. As the valid values for each parameter (i.e. the domain restrictions of the parameter) may not be available, the user is allowed to provide additional information about the valid domains (including for parameters based on complex data types, which are decomposed in a set of individual parameters). Table 2 shows an example of how the tool user should specify the domains for an example web service named *ValidateService* that provides the following operation to the clients: *ValidateObject (String name, String date, String trackingNumber, int number)*.

Two options are available for generating the workload. In the first option (*user-defined workload*), the user should implement a workload emulation tool that can be easily integrated in our testing tool (by performing some simple configurations). To simplify the implementation of the workload generator, there are several easy to use client emulation tools like soapUI [43] that can be used. The second option is to use the *random workload generator* (WG) implemented by the scanner, which is able to generate a workload automatically by performing the following steps:

1. *Generate test values for each input parameter* using the web service definitions mentioned above, the tool generates randomly a set of valid input values (i.e. values in

Table 2 Example of specification of the domains for each parameter

ValidateObject.name	`Type:`	`String`
	`Min Length:`	`3`
	`Max Length:`	`15`
ValidateObject.date	`Type:`	`Date`
	`Format:`	`YYYY/MM/DD`
ValidateObject. trackingNumber	`Type:`	`String`
	`Pattern:`	`\u{2}\d{4}\d{4}\d\u{2}`
ValidateObject.number	`Type:`	`Integer`
	`Min Value:`	`100000000`
	`Max Value:`	`999999999`

the parameter domain specified by the user). The number of test values to be generated is also defined by the user.

2. *Generate test calls for each operation* the tool creates a large set of calls for each operation. This consists of the sum of all combinations of the test values generated for all the parameters. For example, take an operation with 10 parameters (`p`) and 5 test values (`v`) for each parameter. The total number of test calls is 9765625 (i.e. v^p).
3. *Select test calls for each operation* as it may be unfeasible to use a workload based on all the test calls generated (e.g. due to time constraints), the tool is able to randomly select a subset of the calls. It is up to the user to specify the size of this subset, which determines the final size of the workload to be used during the tests.

Note that the main problem of the random workload generation approach is that the representativeness of the web service calls is not guaranteed (although our tool allows using workloads of different sizes and randomly generated values are enough in most cases). Thus, this approach should be used only if the user-defined workload approach is not possible. The tool can be easily extended to include more advanced workload generation approaches (as discussed in Sect. 3.1). After the generation process, the workload injector (WI) executes the workload to gather information about the web service typical responses (i.e. responses obtained without injecting attacks).

4.1.2 Attack emulation

The set of attack types implemented by the IPT-WS tool is based on the compilation of the types used by a large set of scanners (three commercial and two open sources: Acunetix Web Vulnerability Scanner [28], IBM Rational AppScan [27], HP WebInspect [26], Foundstone WSDigger [29], and wsfuzzer [30]). This list was analysed and complemented based on practical experience and using information on injection methods available in the literature (e.g. [19,21,44,45]). The final list includes 137 attack types (see Table 1 for examples and [46] for the complete list).

The attacks generation consists of mutating the workload test calls. In practice, valid values are replaced by malicious values. Obviously, the number of attacks to be performed can be extremely huge. Take, for example, a web service with 3 operations with 5 parameters each and a workload with 25 test calls per operation. Applying all the attack types (137) over all the test calls (25) for every parameter (5) of each operation (3) would end up representing 51375 ($137 \times 25 \times 5 \times 3$) web service executions. Depending on the time available, this may be unfeasible. This way, the tool allows the user to specify the number of test calls from the original workload that should be used for the attack load generation. For this, the original (valid) test calls are ranked based on their potential ability to help in detecting vulnerabilities, and then, a subset is selected.

Ranking is built using the following rules: (1) test calls that led to valid web service responses (i.e. no exception, no server error, and no SOAP error) are in the top of the list; (2) test calls that leaded to web service exceptions are in second place; (3) test calls that leaded to server errors (e.g. http errors in the 400 and 500 intervals) are in third place; and (4) test calls that leaded to client-side errors (e.g. SOAP exceptions) are in the bottom of the list (used only as last resource). It is important to note that test calls that leaded to valid web service responses are placed in the first place as these are the ones with higher probability of exercising the target code in a more effective manner (i.e. with higher coverage). On the other hand, test calls that during the workload execution leaded to exceptions or server errors cannot be simply removed, as they are also useful. For example, test calls that raise web service exceptions due to improper authentication can be used later to check whether an SQL Injection attack is able to circumvent existing authentication mechanisms (details in the next section).

After the generation process, the attacks are submitted to the web service and the responses are collected, providing, together with the information gathered during the

execution of the workload (valid requests and corresponding responses), the support for the vulnerability detection phase.

4.1.3 Vulnerability detection

The last step of the process consists of analysing the responses obtained during the workload and the attacks execution. This is a crucial step to achieve higher coverage and lower false-positive rates, and is done by applying well-defined rules that allow identifying vulnerabilities and excluding potential false positives. In practice, it is based on the following set of steps, which are executed for each of the attacks performed over each parameter:

1. If the response obtained consists of a client-side error (e.g. SOAP stack error), then **the attack was not successful and no vulnerability was explored**, as the web service code was not even executed.
2. Otherwise, if the response is an error then
   (a) If the response for the original test call (before being mutated with malicious values) was the same error, then **the error is not due to the attack**, but to another problem of the service (e.g. software bug or database server problem).
   (b) Otherwise, apply robustness testing over the parameter (as proposed in [42]). Creating a robustness test is very similar to the creation of an attack. However, instead of a malicious input, the attack injector uses a non-malicious invalid input (i.e. values outside the expected input domain). The testing driver module is in charge of asking to the attack injector to generate and execute this request. The tool includes a predefined set of values for each possible parameter domain.
      i. If the responses obtained during robustness testing include the same error, then **the error obtained is due to a robustness problem and not to a vulnerability**. This is the case also of server side validation or input sanitization leads to an error returned to the client.
      ii. Otherwise ***an injection vulnerability exists*** as the attack leaded to invalid responses that could not be observed when using a valid workload or when applying robustness testing. This means that the invalid response is caused by the attack, not by a value out of the parameter's domain. This is a strong symptom of the existence of a vulnerability.
3. Otherwise (i.e. valid response in the presence of the attack), if the execution of the original test call (before being mutated with malicious values) leaded to a database error, server error, or web service exception, then ***an injection vulnerability has been detected*** as the attack was able to exercise parts of the service that were not possible to execute when using the valid workload. An example of this situation is when an attack is able to circumvent an authentication mechanism that was preventing valid test calls from proceeding.
4. Otherwise, if the response obtained in the presence of the attack is the opposite of the response obtained for the original workload call (before being mutated with malicious values), then ***an injection vulnerability has been detected*** as the attack leaded to the successful execution of the operation, which was not the case when using the valid workload. Take, for example, an operation that performs a database modification and only returns a value indicating the success or nonsuccess of the modification. If and attack is able to circumvent an authentication mechanism that was preventing valid test calls from proceeding, then there is a security vulnerability.
5. Otherwise, **no vulnerability** was found. The attack did not change the web service behaviour in a visible manner. It may leave undetected cases where the service does not report relevant information for the detection.

The most critical and difficult step of the algorithm is step 4. In fact, it is quite difficult to establish whether a response obtained in the presence of an attack is the opposite of the response obtained for the original workload call (i.e. before mutation). The rules used for such checking must be simple and should be applied to data types that allow determining the opposition between the result of a valid request and an attack. We propose the set of rules presented in Table 3 to make these decisions. Although these rules may be a source of false positives, because the opposite values are not unique and sometimes depend on the specifics of the application, we can easily integrate new ones and change existing ones. An example of a new rule is to define `{"Successful request"|"Error message!"}` for a *String* parameter that contains messages values. Another example for the rule `{0|1}` that originates false positives for *Int* return codes where `-1` means error, replacing it by the rule `{-1|1}` avoiding the false positive.

4.2 Attack signatures and interface monitoring (`Sign-WS`)

In this section, we design an automatic approach that uses attack signatures and interface monitoring for the detection of injection vulnerabilities (the target is Scenario 2 defined in Sect. 3.3). The goal is to improve the penetration testing process by providing enhanced visibility, yet without needing to access or modify the web services code. The key assumption is that most injection attacks manifest, in some way, in the interfaces between the attacked web service and other

Table 3 Rules for the analysis of opposite responses

Original response	Opposite response
False	True
Fail	Success
0	1
Empty list or array	List or array with values
0	GUID data type
Error	No Error
Invalid	Valid
NOK	OK

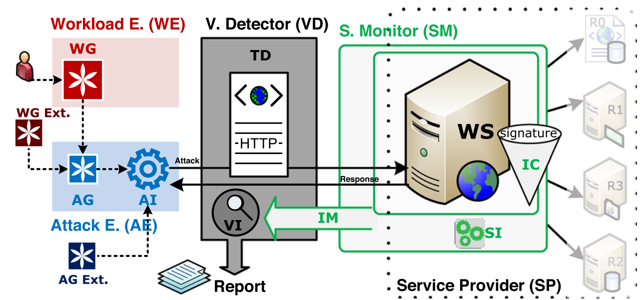

Fig. 6 Sign-WS overall design. (see Fig. 2 for module's names)

```
Select n from t where dsc LIKE '\%input' SIGNATURE\%';
```

(a)

```
Select n from t where dsc LIKE '\%input SIGNATURE\%';
```

(b)

Fig. 7 Examples of queries with signatures. **a** the signature is active; **b** the signature is inside a literal string, and it is inactive

systems (e.g. database, operating system, gateways) and services (e.g. other web services in a service composition). For example, a successful SQL Injection attack leads the service to send malicious SQL queries to the database. Thus, these attacks can be observed in the SQL interface between the service and the database server. The same happens for XPath Injection [19] (XPath or XQuery at the interface with XML files), OS Command Injection [19] (at the interface with the operation system), etc. In practice, the approach targets the types of vulnerabilities that allow web services to be used as front end for attacking backbone resources.

Comparing with traditional penetration testing and with the approach presented in Sect. 4.1, the Sign-WS tool is more effective as it uses additional information that allows increasing the number of vulnerabilities detected and reduce false positives. For example, blind injection vulnerabilities (that exist when an application is vulnerable, but the results of an attack are not visible to the attacker) cannot be detected by traditional penetration testing (or by our IPT-WS tool), but they can be detected by the Sign-WS approach as they can be observed in the interface between the web services and the resource targeted by the attack. On the other hand, detecting vulnerabilities based on the effects of attacks (e.g. changes in SQL queries) is much more precise than considering only the analysis of the web service output.

Figure 6 presents the design of the tool that is detailed in the next subsections. Workload emulation is not discussed as it is based on the random workload generator presented in Sect. 4.1.1 and workload execution is not required.

4.2.1 Attacks emulation

Sign-WS supports two options for generating attacks: it allows the integration of an external tool (e.g. another vulnerability scanner), but also includes a specific generation module similar to the one presented in Sect. 4.1.2. A key aspect in both cases is that signatures are added to the attacks to later support the detection of vulnerabilities. The next subsections introduce the concept of attack signatures and describe how they are added to the attacks when the internal attack generator is used and when an external tool is applied. The following paragraphs explain how to define effective attack signatures and how to use them to generate signed attacks or sign attacks from an external tool.

Defining Effective Attack Signatures In the literature, attack signatures are defined in multiple ways, depending on the type of system studied, but according to [47] an attack signature is "a distinctive complex pattern used to detect system penetration, which may involve comparison of audit and log data from a variety of sources within the computing platform or infrastructure".

In the context of this work, an attack signature is a token that is introduced inside a malicious string (the injection attack) in such way that, if the attack is successful, the token is observable somewhere in the interfaces of the service. For example, in a successful SQL Injection attack [15], the signature should show up in the manipulated SQL command (the target of the attack), outside any literal string (i.e. as a part of the actual command), revealing that it is possible for attackers to modify the structure of the command sent to the database server. In this case, the signature is considered active (see example in Fig. 7a). Otherwise, if the signature is placed inside a literal string, it is considered inactive and is inoffensive (Fig. 7b).

Defining attack signatures is not easy. On one hand, signing attacks with complex signatures may not be possible due to length limitations, restrictions in terms of the characters that can be used, etc. On the other hand, very simple signatures may raise false positives, as there is the risk of using a signature that matches a valid keyword (the valid keyword would wrongly suggest the presence of the signature). This

`_12_p` **(a)** `_21_o` **(b)**

Fig. 8 Signature token used: **a** regular token; **b** reversed

Table 4 Examples of signed SQL Injection attack types

Signed SQL Injection attack	
(1)	`' %SIGNATURE%`
(2)	`\' %SIGNATURE%`
(3)	`'-- %SIGNATURE%`
(4)	`' or 0=0 -- %SIGNATURE%`
(5)	`%WORKLOAD%'%SIGNATURE%`
(6)	`%WORKL_%'or %SIGNATURE%=%SIGNATURE% --`

way, to maximize the success of the approach, the signature token must be: **unambiguous**—the signature must not be easily confused with the tokens/keywords regularly found in the context of the applications being tested; **inoffensive**—the signature must not include characters that may be filtered, escaped, or refused. Although the goal is the attack to pass the protection mechanisms, the signature token must be harmless; **informative**—the signature must include information about what is being attacked to later allow the identification of the vulnerable input; **short**—the token must be as short as possible to avoid problems with limited length fields or protection mechanisms as length validators, which are extremely common in web services.

The proposed signature model is composed of a set of five elements, including two delimiters, two identifiers (that represent the information transported by the signature), and a qualifier (see example in Fig. 8). The first delimiter ('_') marks the beginning of the signature, the first identifier represents the web service operation or resource being tested, the second identifier is the input parameter attacked, and the second delimiter ('_') marks the end of the signature information. The qualifier, placed after the second delimiter, indicates whether the signature is applied in normal or reversed mode, as explained below.

This model allows short and informative enough signatures. To reinforce unambiguity, for each attack a confirmation request is submitted, containing a reversed version of the signature. This is important to decrease the probability of using a signature that, by coincidence, partially matches a part of the target command, also providing a second validation of the vulnerability. As signatures do not include any "special" characters, they do not suffer any transformation due to existing escaping routines, thus assuring inoffensiveness. Obviously, to guarantee portability and allow adapting to different types of web services, the user of the tool is allowed to configure the signature model he wants to apply (using regular expressions).

Figure 8 shows the signature model (including the reversed version) used by our tool. When building the signature, digit '1' is replaced by the identifier of the web service operation under testing, and digit '2' is replaced by the identifier of the input parameter attacked. Each identifier can be a number (10) or a letter (52). The attack injector maintains a dictionary of the meaning of each identifier. If the number of operations or input parameters is greater than 62 (although it rarely is), then it will be necessary to add digits to the signature model described in configuration files, increasing the size of the signature.

Generating Signed Attacks As presented in Sect. 4.1.2, the approach implemented by the internal attack generator (AG) consists of mutating the workload requests. In practice, valid values are replaced by the malicious values. However, in the Sign-WS tool signatures are added to the attacks (by the attack injector component) for later supporting vulnerability identification. Table 4 presents some examples of SQL Injection signed attacks. The meaning of each of the tokens (added by the attack generator) is as follows:

- `%SIGNATURE%` is a placeholder to be replaced by the signature token dynamically generated. By using this token, the tool user can control the specific location of the signature inside the malicious string;
- `%WORKLOAD%` is a placeholder to be replaced at runtime by the value of the input in the original workload request. It is useful because it helps disguising the attack with valid data, avoiding some validators that perform pattern matching. If the workload was correctly generated to fit this kind of domain restrictions, using it as a base to generate attacks increases the chances of generating a successful attack.
- `%WORKL_%` similar to `%WORKLOAD%`, but in this case only the initial characters are used in order to maintain the total length of the input. This helps avoiding length validators. For instance, considering a validator that accepts a string with a length between 100 and 150 characters (or even a more restrictive range), if the workload request is valid, then the attack generated using this token is also valid.

The defined attacks cover the majority of the cases, using techniques that try, for instance, to avoid weak escaping mechanisms by combining multiple escaping characters together. However, the user can easily add more attacks using the rules defined above. A key aspect is that, to reveal a vulnerability, the attack does not need to be successful on accessing, modifying, or destroying data. For example, in the case of SQL Injection what is required is the attack to be able to change the structure of the SQL query in such way that the signature token can be identified in the service inter-

faces as being active. The same is valid for other types of injection vulnerabilities.

Signing Attacks For supporting the integration of an external tool to generate attacks, the attack injector intercepts the requests performed by that testing tool. In this case, the attack injector was developed in such way that all the requests performed by the testing tool are intercepted, locally stored, and finally forwarded (without any change) to the target web service. In this phase, the requests are not signed, as the goal is to allow the penetration tester to perform its work without any interference (except a very small delay introduced by the attack injector; however timing issues are not particularly important especially during testing).

When the testing tool completes the testing process, all the stored requests are analysed by the attack injector, searching for attacks (i.e. requests containing malicious input strings). When an attack is found, a signature token is added to the attack string. The signature includes information about the component and input under attack (obtained from the original request) that is also inserted in the dictionary that maps the identifiers to the inputs they represent. The place in the attack string where the signature is inserted can be defined by the user of the tool, in the form of key characters. These **key characters** depend on the type of attack being conducted; for example, in our experiments with SQL Injection we considered the typical characters necessary to launch SQL Injection campaigns: literal string delimiters ('and "), the equal character (=) often used to manipulate SQL conditions, and the parenthesis characters () and (), used to manipulate or add subqueries. After this, the new request is sent to the application. If no key character is found, then the request is discarded, as it is not considered an attack.

4.2.2 Service monitoring

Simultaneously to the submission of the attacks containing the attack signatures, it is necessary to monitor the interfaces of the application to capture the executed commands (IC in Fig. 6). Depending on the type of interface, there are multiple options to monitor web applications, including use network packet sniffing, use a proxy, and instrument the code. In the particular case of Sign-WS, we use a particular case of the later option: we perform driver instrumentation. In practice, when the interface to be monitored is accessed through a driver (e.g. Java applications use JDBC drivers to access the database server), this driver can be instrumented to include monitoring facilities. Obviously, driver instrumentation is an intrusive technique, but the modifications can be done outside the core of the applications being tested. In fact, it is possible to create an instrumented version of a specific driver that can even be used in different applications. Even so, one must be extremely careful in order not to introduce bugs in the instrumented driver during this process.

For example, to allow the Sign-WS tool to monitor the queries issued to the database we instrumented the JDBC driver using aspect-oriented programming (AOP) [48]. AOP is a well-known programming paradigm that allows injecting crosscutting concerns into any application in a non-intrusive way [48]. The Java Database Connectivity (JDBC) API is designed to access any kind of tabular data, but it is mostly used when a Java application needs to manipulate data stored in a relational database [49]. It is the responsibility of each database vendor to provide a library containing the JDBC driver and the implementation of the JDBC API for Java applications to interact with its database management system (DBMS). AOP was used to transparently intercept the key points inside the JDBC library where the SQL commands are sent to the database server. The result of this process was a new driver library. To use this driver during the web application testing process, what is needed is to refer the modified version in the classpath of the application instead of the original one. During tests execution, this enhanced driver library executes, without harming the normal behaviour of the driver, the examination and processing of the queries, looking for the presence of the attack signatures. A similar approach is used for other drivers.

When the signed attacks are submitted to the service, the information collector (IC) gathers information from the web service interfaces. This information is stored by the information manager (IM) in a database (together with the corresponding requests) by the information manager and later used by the vulnerability detector (VD) to identify vulnerabilities.

4.2.3 Vulnerability detection

After capturing the commands at the service interfaces, it is necessary to process and analyse them to detect potential vulnerabilities. This way, when a signature token is found outside a literal string in a command sent to an external resource, this means that there is a vulnerability in the web service. Thus, before applying regular expressions to find signatures, it is necessary to process the data in order to remove the inoffensive parts (e.g. control characters, well-formed literal strings).

Figure 9 shows an example of the transformations applied to SQL queries during the command processing steps (a similar approach is used for other types of commands). In *step 1*, all the correctly escaped slashes (`\`), apostrophes (`'`) and quotes (`"`) are removed from the string (obviously, the definition of "correctly escaped" varies according to the type of commands that are being processed). In *step 2*, the remaining literal strings identified by the regular expression "`'[^']*'`" are removed, with this regular expression

```
1: Select n from t where dsc LIKE '%input' _28_p%';
2: Select n from t where dsc LIKE '%input' _28_p%';
3: Select n from t where dsc LIKE _28_p%';
```

(a)

```
1: Select n from t where dsc LIKE '%input\' _28_p%';
2: Select n from t where dsc LIKE '%input _28_p%';
3: Select n from t where dsc LIKE ;
```

(a)

Fig. 9 Command processing steps demonstrated with SQL queries. The lines show the transformations applied to each query during the processing. Tokens in ~~strikethrough~~ are removed in that step. Tokens in *italic* are active signatures. **a** An active signature is found; **b** no active signature is found

representing a collection of characters delimited by two apostrophes that cannot contain apostrophes inside. Any attack signature that still remains in the command after this process (*step 3*) is considered active, as can be observed in Fig. 9a. In this case, a vulnerability is identified, having the associated information: '2' (operation or resource under testing) and '8' (input parameter attacked). This is the information that allows linking the vulnerability to the input than can be used to exploit it. Obviously, the attack signatures should include the information needed to make a correspondence between this information and the inputs of the application under testing. An alternative technique would be to use a query parser to parse the SQL. However, the query parser needs more time to execute than processing the regular expressions. Even worse is that the parser usually requires well- formed queries and the signatures used most of the times lead to incorrectly formed queries that will force the parser into unexpected cases that may take more time to process and introduce bigger performance impacts.

4.3 Runtime anomaly detection (`RAD-WS`)

In this section, we present an automatic approach for the detection of injection vulnerabilities based on anomaly detection and that includes two main steps (the target is Scenario 3 defined in Sect. 3.3). First, we generate and run a workload to exercise the web service under testing and learn the profile of the internal commands issued (e.g. SQL and XPath commands). Afterwards, we apply a set of injection attacks and observe the internal commands being executed. This allows us to detect existing vulnerabilities by matching commands during attacks with the valid set of commands previously learned.

Comparing to the approaches proposed before, RAD-WS takes advantage of information about the internal behaviour of the service under testing, which allows increasing detection coverage. Additionally, it allows relating the attacked inputs with the anomalies (i.e. the mismatches between commands during attacks and the valid set of commands previously learned) and locates the vulnerabilities in the code

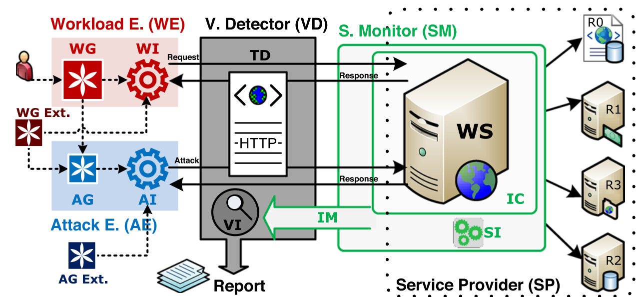

Fig. 10 RAD-WS overall design. (see Fig. 2 for module's names)

of the web service. Figure 10 presents the design of the tool components, which are detailed in the next subsections. The attack emulator component is not presented as RAD-WS uses the one implemented by IPT-WS (presented in Sect. 4.1.2).

4.3.1 Workload emulation

The workload generator (WG) module is based on the random workload generator presented in Sect. 4.1.1. To learn the commands, the workload injector exercises the service by executing the generated workload. During the workload execution phase, internal commands (e.g. SQL and XPath commands) are intercepted (using AOP, as presented in Sect. 4.3.3) and parsed in order to remove the data variant part (if any) and a hash code is generated to uniquely identify each command. In other words, the information used does not represent the exact command text, since commands may differ slightly in different executions, while keeping the same structure. For example, in the SQL command "`SELECT * from EMP where job like 'CLERK' and SAL > 1000`", the job and the salary in the select criteria (`job like STRING? and sal > NUMBER?`) are dependent on the user's choices. Thus, instead of considering the full command text, we just represent the invariant part of it.

Each hash signature is associated with a source code entry point (which is provided by the AOP framework) in a Map structure. This does not mean that we need the original application's source code; it just means that we need bytecode compiled with source code line information, which is generally the case, even in production applications as it provides extra information on failure events. In the previously referred Map structure, each key corresponds to a given source code point and has a set of associated valid/expected hashed commands. Note that, in a given point there might be several valid commands (this is why we need a set of valid commands for each source code point).

An important aspect is that the workload must guarantee a minimum level of code coverage (as discussed in Sect. 3.1). Although this does not assure a complete learning of internal

commands, it allows us to have a high confidence degree. This way, the tool allows easily integrating an external code coverage analysis tool. Additional workload requests are generated if the coverage value is under a given threshold defined by the user.

4.3.2 Service monitoring

For the web service instrumentation, we use the well-known aspect-oriented programming (AOP) [48] to intercept key web service execution points and introduce the vulnerability detection mechanisms. Vulnerability detection starts by automatically identifying all the locations in the web service code where commands are executed. This is achieved by using AOP to intercept all the calls to methods that belong to APIs used to execute SQL commands (e.g. JDBC, the Spring Framework JDBC), to evaluate XPath expressions (e.g. JAXP, JaxenXPath), etc. Virtually any API can be added, as the only requirement is to know the signature of the method to be intercepted. At runtime, methods that issue SQL commands, XPath queries, etc. are intercepted (using AOP) and delivered to the information collector. It is important to emphasize that instrumentation does not change the service behaviour (the code logic is not modified) and that it is only meant for the RAD-WS tool (it is removed after testing).

4.3.3 Vulnerability detection

To detect vulnerabilities, we perform security checks for each data access executed during the attack phase. All commands (SQL, XPath, etc.) are intercepted (using AOP), hashed, and stored during that phase. These are compared to the values of the learned valid commands for the code point at which the command was submitted. In practice, the matching process consists of looking up the current source code origin in the previously referred Map structure and getting the set of hash codes of the valid (learned) commands for that point. This set (generally quite small) is then searched for an element that exactly matches the hash of the command that is being executed. If a match is not found, the occurrence (i.e. the potential vulnerability) is logged for future reference. The log entry includes a reference to the code location where the vulnerability was detected, the query that was executed in the presence of the attack, and information about the operation input values, namely the attacked parameter and the attack value. If the source code origin is not found in the Map lookup, the log indicates that the line was not learned. This case indicates that the learning phase is incomplete (coverage was not good enough) and that a more exhaustive workload is required. Note that the lines that have not been learned provide indications on how to improve the workload to increase coverage.

5 Evaluation

The evaluation of the tools focuses on the key innovations introduced: **a generic design** and the **modularity** of such design. Modularity allows to easily develop new techniques or improve the available techniques just by adding new modules or improving the existing ones. However, this should be achieved *without reducing the vulnerability detection effectiveness*.

As case study, we used the three tools presented before and three commercial vulnerability scanners to detect SQL Injection vulnerabilities in a set of SOAP web services implemented in Java. In addition to the modularity analysis, the goal is to evaluate the effectiveness of each tool and to understand whether they are an effective alternative to commercial ones. Two well-known metrics were used in this evaluation: **detection coverage** (percentage of existing vulnerabilities detected by the tool) and **false-positive rate** (percentage of vulnerabilities reported but that do not exist).

In practice, the goal is to answer the following questions: (1) does the proposed approach supports building modular tools? (2) do different access conditions to the web services under testing allow more advanced tools to achieve higher effectiveness in terms of coverage and false positives? (3) do the proposed tools have greater effectiveness than existing state-of-the-art vulnerability scanners, thus representing a viable alternative?

Time and resource consumption values are not a major factor when evaluating the tools and their applicability and were left out of scope of this work. However, we can say that the complete testing process had a duration ranging from minutes to few hours depending on the complexity and throughput of the service under testing. This duration is considered acceptable, and its impact decreases when compared with the importance of their detection coverage and false-positive rate: vulnerabilities left undetected have the potential for huge financial losses, while false positives will require that developers allocate effort to fix nonexistent vulnerabilities. In terms of memory, none of the tools required more than 1GB to perform their tasks, meaning that can be supported by any computer nowadays.

5.1 Tools and experimental setup

The experimental evaluation was based on the web services of the "Benchmark for SQL Injection Vulnerability Detection Tools" proposed in [8]. Using a benchmark allows evaluating and comparing our tools according to specific characteristics in a standard way. The benchmark consists of 21 services, with 80 operations with a total of 158 known SQL Injection vulnerabilities. These services have been adapted from three standard benchmarks developed by the Transactions Processing Performance Council, namely TPC-App, TPC-

C, and TPC-W (see details on these benchmarks at [50]). To characterize the vulnerabilities that exist in the web services, we invited a team of 3 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. Details are at [8].

The three commercial scanners used in the experiments are representative of the state of the art in vulnerability testing for web applications and web services: HP WebInspect [26], IBM Rational AppScan [27], and Acunetix Web Vulnerability Scanner [28]. For the results presentation, we have decided not to refer directly to the commercial scanners to assure neutrality and because licenses do not allow, in general, the publication of evaluation results. The three tools are referred in the rest of this paper as VS1, VS2, and VS3 (without any order in particular).

In terms of configurations, our three tools were configured to perform tests using a workload size of 10 elements. This means that for a web service with 5 operations and 7 parameters each operation, the total number of request performed is 49000 ($5 \times 10 \times 7 \times 140$) and the duration of the workload execution is about 15 minutes. For the commercial tools, information about the domain of each web service input parameter was provided when allowed by the tool. If the tool requires the user to set an example invocation per operation, the example respected the input domains of operation. All the tools in this situation used the same example to guarantee a fair evaluation. The tests were performed 3 times for each tool, always starting from the same initial conditions. The tested services are stateless, and the testing tools work in a deterministic way resulting in the same execution flow and the same results in every run. Details on the presented results are available at [46].

5.2 Modularity analysis

From the definition of the approach and components presented presented in Sect. 3, we can identify two main modularity attributes: the elements inside a component should address only one concern and be strongly connected inside themselves **(high cohesion)**; at the same time, a component should not address more than one concern and it should be independent from other components **(loose coupling)**. This means that they should have few connections with other components of the tool.

To build a modular tool, one should start by identifying concerns and separate them, to build software components that address each concern. After this separation, it is possible to hide the complexity of each module behind abstractions and interfaces.

There are many metrics for characterizing the degree of modularization. However, most of those metrics lack some meaning in the values produced in absolute terms. In fact, most metrics are only useful for comparison purposes. Furthermore, it is not a fair assessment to say that a program implementation is more modular than other if the goals and functionalities implemented are different, mainly because *concerns* are different and the number of crosscutting concerns may be higher in one implementation than in the other.

Due to these limitations and since comparison between our implementations and the existing tools would not be fair, we decided to perform a qualitative analysis instead of a quantitative analysis. For that, we analysed the component diagram of our tools in order to identify design patterns that can give us evidence that the components are compliant with the required modularity attributes.

We identified four concerns that are necessary in order to be possible to develop, change, or maintain our tools without knowing or change the entire tool. Those concerns are: workload generation, attack load generation, monitoring, and detection. Each of these concerns was isolated within a module, as shown in Sect. 3, guaranteeing a high cohesion among the elements included in the module. Additionally, to guarantee loose coupling between modules, every component provides a well-defined interface to the remaining ones, and this interface does not provide functionality that is not related with the concern that the module addresses.

As shown in Fig. 2, the connectivity between components is through well-defined interfaces. Also, the components do not depend on the remaining ones. These are typical characteristics of a modular design, and the less connection to other component and the less concerns are addressed by one component, the more modular the component under analyses is. Finally, high cohesion (specifically, functional cohesion) can be observed in the external modules as all parts of the module are placed together and share the same goal (i.e. all the elements present on the module address the same concern, and all of them contribute to the same task).

5.3 Overall results for the three tools

To understand whether improved access conditions to the web services under testing allow achieving higher effectiveness, our three tools were executed on top of the web services (in this case, Sign-WS is used only with the internal attacks generator). Figure 11 presents the summary of the results (the first bar in the graph presents the total number of known vulnerabilities).

The different tools reported a different number of vulnerabilities. As expected, RAD-WS identified the higher number of vulnerabilities (≈75 % of the total number of vulnerabilities) and Sign-WS coverage (≈74 %) is very close to RAD-WS. This suggests that the added visibility indeed

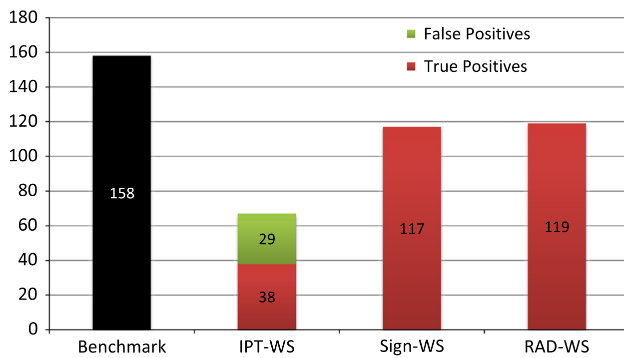

Fig. 11 Summary of the results for the three tools

improves the detection process, but detection based on interface monitoring is also very effective. A very important aspect is that both RAD-WS and Sign-WS reported 0 false positives. IPT-WS is the tool with the lowest detection coverage and the only one reporting false positives (but also the one with less requirements in terms of information). In practice, RAD-WS was able to detect all the vulnerabilities detected by Sign-WS, which in turn detect all the true vulnerabilities reported by IPT-WS. Additionally, the results obtained are equivalent to the ones obtained by the original tools [9–11], with small variations observed due to the non-deterministic nature of the testing processes. However, these variations are always under 0.01 %, which is quite acceptable when comparing the detection performance of tools. The following paragraphs analyse the results in more detail.

IPT-WS detected a low number of vulnerabilities (≈25 %) and reported a large set of false positives (≈43 %). The low coverage and very high rate of false positives can be explained by the fact that this tool is based in the limited information provided by the web services requests and responses. For example, vulnerabilities whose exploitation does not manifest in the output of the service cannot be detected. Furthermore, unclear responses induced the mechanism to report a large number of vulnerabilities that in fact do not exist.

The first observation regarding Sign-WS is that it is able to detect 117 of a total of 158 vulnerabilities (≈74 %), presenting much higher detection coverage than IPT-WS (which detected only 38 real vulnerabilities). As expected, the increased visibility on the web services interfaces, provided by the use of signed attacks and interface monitoring, allows the approach to detect vulnerabilities that would not be possible to detect by analysing only the web services responses. The second observation is related to the fact that the tool did not report false positives. This increases the confidence in the vulnerabilities detected by Sign-WS in future campaigns. As mentioned before, the rare scenario where false positives would manifest is when tokens similar to the signatures (both normal and reversed) are used in the construction of application's queries. That is not the case in this set of services and also in the large majority of applications (anyway, the tool user is able to configure the signature model, which may allow avoiding matching similar keywords). This way, we do believe that these results can be generally reproduced in real-world web services.

The detection methodology of RAD-WS allowed the identification of the highest number of vulnerabilities. The tool has identified 119 vulnerable inputs (a coverage of ≈75 %), two more that Sign-WS, which shows that internal visibility allows detecting vulnerabilities that do not manifest in the web services interfaces. This happens in the rare cases where the signatures are rejected or transformed by a validation mechanism, while the RAD-WS is able to detect differences in the query profile independently from signatures. Examples are fields with a length validator smaller than the signature but big enough for other attacks (e.g. "'– –"), or a field with a pattern validator that the signature does not fit but that another attack may fit. Another important aspect is that the tool did not report any non-existing vulnerabilities (0 false positives).

As mentioned before, the quality of the results obtained is directly related to the coverage of the workload used in profiling phase, as it is during this phase that the valid SQL commands are learned. The results and the detailed analysis of the web services code showed that the workload generated by the tool was sufficient to learn most of the commands in which vulnerabilities exist. The analysis also revealed that the vulnerabilities not detected are due to situations where a vulnerability is preceded by another very similar vulnerability, and so, the second can only be detected after fixing the first. In these cases, also a penetration tester would probably not find the vulnerabilities. However, the attacker may find ways to explore the vulnerable code, and also, the code needs to be fixed or will create problems in the future. This way, we also consider these vulnerabilities for evaluation purposes.

The effectiveness of the workload achieved during the profiling phase of our experiments is also achievable in real-world situations, as the web services developers typically have access to the source and possess enough knowledge about the service to correctly define the input domains. In fact, the web services tested are a good example of real-world complex web services (the complexity of the *TPC* specifications can be attested in [50]) having a huge number of parameters. Although the web services code was not written by the authors of the present paper, it was possible to define a workload that assures a complete learning. This suggests that the process can be reproduced in other real web services scenarios. Nevertheless, we believe that it is important to research automatic workload generation approaches based on the source code analysis in order to achieve higher code coverage and to increase automation (e.g. to eliminate the need for the user to provide input domains).

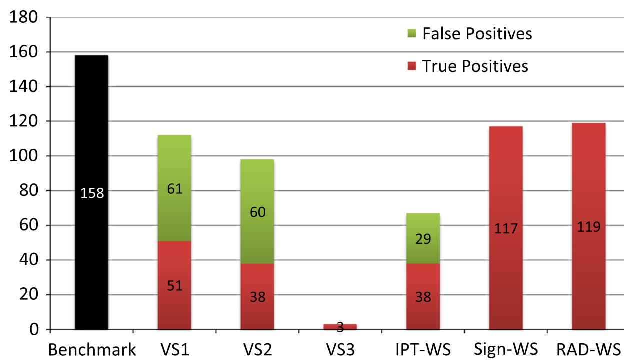

Fig. 12 Comparison between our tools and three commercial vulnerability scanners

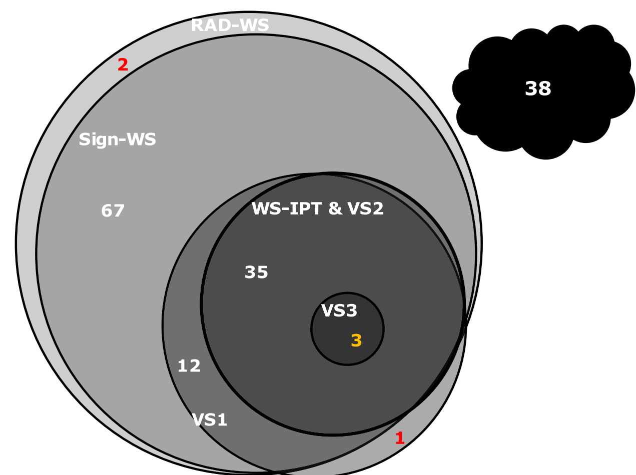

Fig. 13 Intersection of vulnerabilities detected

5.4 Comparison with state-of-the-art vulnerability testing tools

To assess whether the proposed tools represent a viable alternative, we compared their effectiveness with three well-known and widely used commercial vulnerability scanners. Overall results are shown in Fig. 12, with the first bar presenting the total number of vulnerabilities in the code.

Columns 2, 3, and 4 in Fig. 12 show the vulnerabilities reported by the commercial tools. As we can see, the different tools reported a different number of vulnerabilities and the coverage for the commercial tools is always under 35 %. Among these, VS1 identified the higher number of vulnerabilities (≈32 % of the total vulnerabilities). However, it also reports a very higher number of false positives (≈54 %). 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 parts of the code.

Comparing our tools with the commercial scanners, we can observe that both Sign-WS and RAD-WS consistently present better results, in terms of both coverage and false positives, largely outperforming any of the commercial tools. On the other hand, the tool based on improved penetration testing (IPT-WS) presents better results than two of the commercial tools (VS2 and VS3), but presents a lower coverage than VS1 (≈24 % against ≈32 %). However, in terms of false positives IPT-WS performs better than VS1 (≈43 % and ≈54 %, respectively). A detailed analysis of the results and of the web services under testing showed that IPT-WS is better than VS1 on the identification of false positives (i.e. the rules we implement are more precise), but is less effective on exercising the target services (the workload is less effective).

Figure 13 illustrates the analysis of the intersection of the vulnerabilities detected by the different tools. The areas of the circles are roughly proportional to the number of vulnerabilities detected by the respective tool, whose name is indicated close to the circle. The same does not happen with the intersection areas, as it would be impossible to represent it graphically.

As we can see, RAD-WS is the tool that detected the largest number of vulnerabilities, reporting 2 vulnerabilities that no other tool reported. A large amount of vulnerabilities (67) were detected only by RAD-WS and Sign-WS, representing more than 55 % of the total number vulnerabilities reported, showing the advantage of tools with extra information when compared with penetration testing tools. Additionally, only 3 vulnerabilities were detected by all the tools, but this number is obviously limited by the low coverage of VS3. Regarding the penetration testers, VS1 presented the higher detection coverage, as it is able to identify all the vulnerabilities detected by IPT-WS and VS2, plus 13 vulnerabilities. VS1 was also able to identify one vulnerability that no other tool detected. Although none of the tested tools was able to exploit this vulnerability, VS1 uses heuristics to identify vulnerabilities that are in most cases very liberal, reporting vulnerabilities from little evidences that in many cases result in false positives but in this case resulted in a really existing vulnerability.

The final remark is relative to the *38 vulnerabilities that were not detected*. After a manual analysis of the services, similarly to what was discussed in Sect. 5.3 we concluded that: (1) many of those are vulnerabilities located in places in the code hard to reach via black-box testing, and the workloads used are not yet complete enough to be able to execute those code paths, and (2) sometimes a vulnerability is preceded by another very similar vulnerability, and so, the second can only be detected after fixing the first. The solution for the first case is to develop new and better ways for generating the workload and attacks. Regarding the second case, we need to apply an iterative process with alternate

cycles of vulnerability detection and correction. Nevertheless, these topics are outside of the scope of this paper and the upgrade of these modules can be done without impacting the proposed architecture (which shows the advantage of following our generic approach for designing vulnerability testing tools).

6 Threats to validity

There are two points in the work that may threat its validity and that are important to be discussed, as follows.

- It was not possible to gather measures on how easy it is to implement new tools, as large parts of the work were conducted by the authors of the paper on adapting the design of existing tools. We plan to do future experiments with third-party developers that should in the end try to detail the difficulties faced.
- The evaluation includes a single case study that may be limited for understanding the real effectiveness of the tools. However, the main goal of the evaluation is to show that the tools outperform commercial security scanners (similarly to their original versions), while adopting the direct benefits of the designing approach proposed: maintainability and flexibility.

The case study itself has shortcomings that also need to be discussed, as follows:

- The vulnerabilities focused are only SQL Injection vulnerabilities. This may limit the generalization of some of the concepts presented in the paper. However, all these concepts are extensible to injection vulnerabilities in general, which represent the biggest threat in the web services domain.
- The services used are only SOAP-based web services. This may also limit the generalization of the results, but all the approaches presented here are technology independent and may be used to test other kinds of services. The only requirement is having services with a well-defined interface that is understandable by an application.
- The benchmark used for evaluation was proposed by the authors, which may bias the results of the evaluation. However, the benchmark is based on a well-defined approach and its representativeness depends on the services used as workload which in this case come from third-party implementations of widely used performance benchmarks.

7 Conclusions

In this paper, we proposed a standardized and consistent procedure to design vulnerability testing tools. This includes the architecture of such tool, the generic approach, and a set of well-defined components. The approach provides an integrated support for developing innovative and more effective tools whose modularity allows iterative improvements simply by upgrading each module by improved versions of themselves.

The proposed approach was used to design tools *implementing three different techniques* for detecting injection vulnerabilities: improved penetration testing, attack signatures and interface monitoring, and runtime anomaly detection. These tools use different parts of the generic architectures, showing the versatility of the approach. The tools target different testing scenarios (i.e. target services with different access conditions) offering a *multitude of options to the testing teams*.

A benchmark for vulnerability detection tools was used as case study to demonstrate the effectiveness of tools for the particular case of detecting SQL Injection vulnerabilities. The experimental evaluation showed that the tools can effectively be used in different scenarios and that they outperform well-known commercial tools by achieving higher detection coverage and lower false-positive rates.

Future work includes implementing vulnerability detection tools targeting other types of vulnerabilities and services. Based on the proposed approach, we will develop a new integrated and modular tool to allow developers to implement new detection techniques or improve the existing just by integrating new modules. During this process, we will evaluate our approach using external developers. The new tool will also include the implementation of a framework for detection of vulnerabilities in service-based infrastructures.

Additionally, it is necessary to improve the modules where the tools performed less effectively as in generating more effective workloads and studying the code coverage during test execution. Finally, the efficiency of the developed tools in terms of performance and scalability need also to be evaluated as they impact on the software usability.

Acknowledgements This work has been partially supported by the project *CErtification of CRItical Systems* (www.cecris-project.eu, CECRIS), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324334, within the context of the EU Seventh Framework Programme (FP7).

References

1. Alonso, G.: Web Services: Concepts, Architectures and Applications. Springer Verlag, Berlin (2004)

2. Christey, S., Martin, R.A.: Vulnerability type distributions in CVE, V1. 0 **10**, 04 (2006)
3. Zanero, S., Carettoni, L., Zanchetta, M.: Automatic Detection of Web Application Security Flaws, Black Hat Briefings (2005)
4. Vieira, M., Antunes, N., Madeira, H.: Using Web Security Scanners to Detect Vulnerabilities in Web Services. In: IEEE/IFIP International Conference on Dependable Systems & Networks, DSN'09. (Estoril, Lisbon, Portugal, 2009), pp. 566–571 (2009). doi:10.1109/DSN.2009.5270294
5. Council, T.P.P.: TPC BenchmarkTM App (application server) Standard Specification, Version 1.3. http://www.tpc.org/tpc_app/ (2008)
6. Meier, W.: Web, Web-Services, and Database Systems. In: Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R. (ed.) No. 2593 in Lecture Notes in Computer Science, pp. 169–183. Springer, Berlin Heidelberg (2003)
7. Fonseca, J., Vieira, M., Madeira, H.: Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. In: 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007) (Melbourne, Australia, 2007), pp. 365–372 (2007). doi:10.1109/PRDC.2007.55
8. Antunes, N., Vieira, M.: Benchmarking Vulnerability Detection Tools for Web Services. In: IEEE Eighth International Conference on Web Services (ICWS 2010) (Miami, Florida, 2010), pp. 203–210 (2010). doi:10.1109/ICWS.2010.76
9. Antunes, N., Vieira, M.: Detecting SQL Injection Vulnerabilities in Web Services. In: Fourth Latin-American Symposium on Dependable Computing 2009 (LADC '09), pp. 17–24. IEEE Computer Society, Joao Pessoa, Brazil (2009). doi:10.1109/LADC.2009.21
10. Antunes, N., Vieira, M.: Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services. In: 2011 IEEE International Conference on Services Computing (SCC) (IEEE, 2011), pp. 104–111 (2011). doi:10.1109/SCC.2011.67
11. Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H.: Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services. In: 2009 IEEE International Conference on Services Computing (SCC 2009) (Bangalore, India, 2009), pp. 260–267 (2009). doi:10.1109/SCC.2009.23
12. Chappell, D.A., Jewell, T.: Java Web Services. O'Reilly & Associates Inc, Sebastopol (2002)
13. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Service Definition Language (WSDL) 1.1. http://www.w3.org/TR/wsdl (2001)
14. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Inc, Sebastopol (2007)
15. OWASP Foundation, OWASP top 10 2013. Tech. rep., Open Web Application Security Project (2013)
16. Foundation, O.: Open Web Application Security Project. http://www.owasp.org/ (2001)
17. Acunetix. 70 % of Websites at Immediate Risk of Being Hacked! http://www.acunetix.com/news/security-audit-results.htm (2007)
18. NTA Monitor, Annual Web Application Security Report. Tech. rep. (2011)
19. Stuttard, D., Pinto, M.: The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. Wiley, Hoboken (2007)
20. Fogie, S., et al.: XSS Attacks: Cross Site Scripting Exploits and Defense. Syngress Publishing, Burlington (2007)
21. Jensen, M., Gruschka, N., Herkenhoner, R., Luttenberger, N.: SOA and Web Services: New Technologies, New Standards—New Attacks. In: Fifth European Conference on Web Services. ECOWS '07, pp. 35–44 (2007)
22. OWASP Testing Project: Testing for web services—OWASP testing guide v3. Tech. rep, Open Web Application Security Project (2008)
23. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the Art: Automated Black-box Web Application Vulnerability Testing. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 332–345 (2010)
24. I.C.S.S.S.E.S. Committee, 1012-2012—IEEE Standard for System and Software Verification and Validation, IEEE standard 1012-2012 edn. (IEEE Computer Society)
25. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, Hoboken (2011)
26. HP. HP WebInspect. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200 (2008)
27. IBM. IBM Rational AppScan. http://www-01.ibm.com/software/awdtools/appscan/ (2008)
28. Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/vulnerability-scanner/ (2008)
29. I. Foundstone. Foundstone WSDigger. http://www.foundstone.com/us/resources/proddesc/wsdigger.htm (2005)
30. OWASP Foundation. OWASP WSFuzzer Project. http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project (2008)
31. Huang, Y., Huang, S., Lin, T., Tsai, C.: Web Application Security Assessment by Fault Injection and Behavior Monitoring. In: Proceedings of the 12th International Conference on World Wide Web (ACM, Budapest, Hungary, 2003), pp. 148–159 (2003)
32. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: SecuBat: A Web Vulnerability Scanner. In: Proceedings of the 15th International Conference on World Wide Web (ACM, New York, NY, 2006), p. 247256 (2006). doi:10.1145/1135777.1135817
33. Doup, A., Cova, M., Vigna, G.: In: Detection of Intrusions and Malware, and Vulnerability Assessment. no. 6201 in Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2010), pp. 111–131 (2010)
34. Doliner, M.: Cobertura. http://cobertura.sourceforge.net/ (2006)
35. Atlassian. Clover—Code Coverage for Java. http://www.atlassian.com/software/clover/ (2010)
36. Balzarotti, D., et al.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: IEEE Symposium on Security and Privacy. SP 2008, **66**, pp. 387–401 (2008). doi:10.1109/SP.2008.22
37. Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications, In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, **41** (ACM, New York, NY, 2006), POPL '06, p. 372382 (2006). doi:10.1145/1111037.1111070
38. Halfond, W., Orso, A.: AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks, In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, p. 183 (2005)
39. Laranjeiro, N., Vieira, M., Madeira, H.: A Technique for Deploying Robust Web Services. IEEE Transactions on Services Computing PP(99), 1 (2012). doi:10.1109/TSC.2012.39
40. Kaner, C.: Software Negligence and Testing Coverage. In: Proceedings of STAR 96: The Fifth International Conference on Software Testing Analysis and Review (Orlando, FL, 1996), pp. 299–327 (1996)
41. Kindy, D., Pathan, A.S.: A Survey on SQL Injection: Vulnerabilities, Attacks, and Prevention Techniques. In: 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE), pp. 468–471 (2011). doi:10.1109/ISCE.2011.5973873
42. Vieira, M., Laranjeiro, N., Madeira, H.: Assessing Robustness of Web-services Infrastructures. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN'07, pp. 131–136 (2007)
43. eviware. soapUI. http://www.soapui.org/ (2008)

44. Shema, M.: Seven Deadliest Web Application Attacks. Syngress, Burlington (2010)
45. Halfond, W.G., Viegas, J., Orso, A.: A Classification of SQL-injection Attacks and Countermeasures. In: International Symposium on Secure Software Engineering (2006)
46. Antunes, N., Vieira, M.: Vulnerability Testing Tools for Web Services. http://eden.dei.uc.pt/~mvieira/ (2013)
47. Sabhnani, M., Serpen, G.: Why Machine Learning Algorithms Fail in Misuse Detection on KDD Intrusion Detection Data Set. Intelligent Data Analysis **8**(4), 403–415 (2004)
48. Kiczales, G.J., et al.: Aspect-oriented programming. US Patent 6,467,086 (2002)
49. Reese, G., Oram, A.: Database Programming with JDBC and JAVA. O'Reilly & Associates, Inc., Sebastopol (2000)
50. Transaction Processing Performance Council. Transaction processing performance council. http://www.tpc.org/ (2009)

Nuno Antunes received the PhD degree in Information Science and Technology from the University of Coimbra, Portugal. He is an Assistant Professor at the University of Coimbra, Portugal. His interests include the development of dependable and secure web applications and services, experimental dependability and security evaluation, 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 Associate 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.