Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-024-00518-0

ORIGINAL PAPER

Sniping at web applications to discover input-handling vulnerabilities

Ciro Brandi[1] · Gaetano Perrone[1] · Simon Pietro Romano[1]

Received: 19 April 2023 / Accepted: 21 February 2024
© The Author(s) 2024

Abstract

Web applications play a crucial role in modern businesses, offering various services and often exposing sensitive data that can be enticing to attackers. As a result, there is a growing interest in finding innovative approaches for discovering vulnerabilities in web applications. In the evolving landscape of web security, the realm of fuzz testing has garnered substantial attention for its effectiveness in identifying vulnerabilities. However, existing literature has often underemphasized the nuances of web-centric fuzzing methodologies. This article presents a comprehensive exploration of fuzzing techniques specifically tailored to web applications, addressing the gap in the current research. Our work presents a holistic perspective on web-centric fuzzing, introduces a modular architecture that improves fuzzing effectiveness, demonstrates the reusability of certain fuzzing steps, and offers an open-source software package for the broader security community. By addressing these key contributions, we aim to facilitate advancements in web application security, empower researchers to explore new fuzzing techniques, and ultimately enhance the overall cybersecurity landscape.

Keywords Fuzzing · Web applications · Web penetration testing · Rule-based systems · Web security testing

1 Introduction

Web applications are the cornerstone of modern digital interactions, facilitating e-commerce, social networking, and information sharing. However, their widespread use also makes them prime targets for malicious actors seeking to exploit vulnerabilities. The rapid evolution of web technologies and the proliferation of complex web applications have rendered the task of securing them increasingly challenging. One potent weapon in the arsenal of cybersecurity professionals is fuzz testing, commonly known as fuzzing. Fuzzing involves the automated injection of various inputs, often malformed or unexpected, into a target system to uncover vulnerabilities. While fuzzing has proven remarkably effective in identifying security weaknesses, its application to web environments presents unique challenges that have been underrepresented in the existing literature. This article aims to bridge this gap by presenting a comprehensive exploration of fuzzing techniques tailored explicitly to web applications. We delve into the intricacies of web-centric fuzzing, highlighting the idiosyncrasies that set it apart from traditional fuzzing approaches. Our research investigates the complexities of web technologies, including input validation, session management, and client-side interactions, shedding light on the specific challenges faced by security practitioners in this domain.

Furthermore, we introduce a modular architecture for fuzzing that not only enhances vulnerability discovery but also replicates the decision-making processes of security experts. This architecture allows for the customization of fuzzing strategies, enabling users to select and deploy specific modules according to their testing objectives. A *Man In The Middle Proxy* module stores interactions performed to analyze the target application. A *Repeater* module converts the collected interactions into template-based requests that can be used to initiate a fuzzing session. At the end of the fuzzing session, an *Analyzer* module converts the fuzzing results into a formal representation of the conditions observed by a security expert and is used to verify the presence of a vulnerability. We formalize these conditions by introducing the "analyzer observations" concept and show that it is pos-

✉ Simon Pietro Romano
spromano@unina.it

Ciro Brandi
ciro.brandi@studenti.unina.it

Gaetano Perrone
gaetano.perrone@unina.it

[1] Department of Electrical Engineering and Information Technology, University of Napoli Federico II, Naples, Campania, Italy

Published online: 12 April 2024

sible to incorporate them into a knowledge base developed using logic programming. To this purpose, we leverage the declarative semantic of Prolog to implement a vulnerability knowledge base that converts the analyzer observations into input-handling vulnerabilities.

We demonstrate that our proposed architecture facilitates the reuse of certain fuzzing steps, significantly streamlining the security testing process. By reducing redundancy, our approach conserves computational resources and accelerates the identification of vulnerabilities.

To foster collaboration and innovation within the security community, we provide an open-source implementation of our software.[1] This open approach encourages researchers to explore new techniques by modifying individual modules, ultimately promoting advancements in web application security.

In conclusion, our work addresses the unique challenges of web-centric fuzzing, introduces an adaptable modular architecture, showcases the reusability of specific fuzzing steps, and offers an open-source software package. Through these contributions, we aim to enhance the security of web applications, empower researchers to innovate in the field of fuzz testing, and fortify the cybersecurity landscape as a whole.

The remainder of this paper is organized into eight sections. Section 2 introduces the security testing process and describes the input-handling vulnerabilities analyzed in this work. To understand the inner details of the proposed solution, Sect. 3 provides an introduction to Prolog and logic programming. Section 4 analyzes the state of the art with respect to Fuzzing. In Sect. 5, we show the design and implementation of our rule-based fuzzer by delving into the details of its main building modules. The proposed approach is evaluated in Sect. 6 by means of a comparative analysis with the renowned Zed Attack Proxy open-source web application scanner. In Sect. 7, we discuss a few interesting optimization strategies aimed at enhancing the performance of our fuzzer. Section 8 summarizes the obtained results and gives an insight into the prospective evolution of the proposed work.

2 Vulnerabilities in web applications

There is some debate in the literature about the "boundaries" between web security testing and web application penetration testing. Some authors define security testing as the application of automatic approaches to discover vulnerabilities instead of manual methods called penetration tests [1]. A recent study [2] provides a systematic mapping of security testing approaches in the literature by extending the scope to penetration testing methodologies, such as OWASP [3]. According to the semantics proposed by the study, even security testing processes can use automatic, manual, and semi-automatic techniques.

In the business world, the distinction between security testing and penetration testing lies in their respective execution stages. Security testing is employed throughout various phases of the development process, ensuring that security measures are incorporated from the early stages. On the other hand, penetration testing is specifically conducted in the production phase, allowing for testing in a real environment with real-world configurations. Throughout the rest of this paper, we will use the term *Web Application Penetration Testing* (WAPT) to refer to an "opaque-box semi-automatic security testing methodology". This approach combines both automated and manual techniques to comprehensively identify vulnerabilities in web applications.

In this section, we describe a Web application penetration testing process by also briefly illustrating three input-handling vulnerabilities studied in this work, namely, SQL injection, path traversal, and cross-site scripting.

2.1 Web application penetration testing process

Web Application Penetration Testing (WAPT) is an "opaque-box" process that enables companies to discover vulnerabilities in web applications by simulating hacker activities. Here, the "opaque-box" refers to the tester having very little prior knowledge about the target system. The process is composed of two main phases, as described in the following:

1. The penetration tester attempts to create a "footprint" of the web application. This includes:
   - Gathering its visible content by exploring public resources as well as discovering information that seems to be hidden.
   - Analyzing the application and identifying its core functions, especially those for which the web application was designed. The purpose is to have a map of all the possible Data Entry Points that the application exposes, which are the main potential flaws that a hacker recognizes in the target application.
2. In the second phase, the penetration tester knows which path to take and whether to focus on the way the application handles inputs or on probable flaws in its inner logic. To have a comprehensive understanding of all possible application "holes" it is, however, important to explore each of the following areas:
   - Focusing on the application logic means studying the client-side controls to find a way to bypass them.
   - The stage that involves analyzing how the application handles access to private functionality might be the

[1] https://github.com/NS-unina/Rule-Based-Fuzzer.

one to focus on immediately because authentication and session management techniques usually suffer from a number of design and implementation flaws.
- Input Handling attacking techniques are definitely the most widely deployed since important categories of vulnerabilities are triggered by unexpected user input. The application can be probed by fuzzing the parameters passed in a request. See the next section for insights on this topic.
- The website can represent an entry point that allows the attacker to have a complete understanding of the target network infrastructure. Defects and oversights within an application's architecture can often enable the tester to escalate an attack, moving from one component to another to eventually compromise the entire application.

2.2 SQL injection

An SQL injection (SQLi) vulnerability allows a malicious user to manipulate the queries issued by a web application to a back-end database. Exploiting the vulnerability exposes sensitive data that cannot be retrieved when the application is used in an ordinary way. In many cases, a malicious user can modify or delete data, thus causing permanent changes to the application and altering its behavior. In some extreme situations, an attacker can also escalate privileges and compromise the underlying server or other back-end infrastructures.

Union attack

When SQL query results are returned as part of application responses, a malicious user can retrieve data from database tables. This technique makes use of an operator called `UNION`. The `UNION` operator allows the joining of several data types through two `SELECT` statements. It is used to perform an additional SQL query that will show illegitimate data.

Blind SQL

Blind SQL is a type of SQL Injection attack that injects either true or false conditions into the underlying database to verify the presence of a vulnerability through the observation of differences in the application's responses. The attack is often used when the web application is vulnerable to SQL injection. Still, it only displays generic error messages, and it is not possible to confirm the vulnerability by simply observing the response content. Blind vulnerabilities can be used to access unauthorized data through challenging exploitation techniques.

2.3 Path traversal

Path Traversal (PT) is a web application vulnerability that allows attackers to access files and directories outside the web application's root directory. The attack is performed by injecting payloads that allow access to files through directory traversal sequences using special characters (e.g., "../") or by injecting absolute file paths. By exploiting this vulnerability, it is possible to access arbitrary files and directories in the file system of the server, including application source code and critical system configuration files. This attack is also known as "point-blank", "directory traversal", "directory climbing", and "backtracking" [4].

2.4 Cross-site scripting

A Cross-site scripting (XSS) vulnerability may occur when a web application does not validate a client-side code in the input sent through an attribute of an HTTP request. When the input is received and bounced back to the browser as part of the response, the client-side code is executed. Typically, attackers exploit such a vulnerability to send malicious URLs containing malicious client-side code to a target user. Since the target user could already be authenticated in the vulnerable web application context, the injected malicious code might potentially read their cookies, session token, or other sensitive information. Cross-site scripting vulnerabilities can indeed be either reflected or stored. As already anticipated, with reflected cross-site scripting, the web application bounces back the injected payload in the content of the HTTP response. The vulnerability is usually found in search pages, error messages, and whenever the web application needs to send back information obtained in the received request. With stored XSS, instead, the injected code is stored in the web application itself. When the users visit a specific page of the vulnerable application, the maliciously stored content is retrieved, and the code is executed, thus potentially impacting all the users of the application.

3 Prolog and first-order logic programming

Our rule-based sniper discovers vulnerabilities by leveraging a vulnerability knowledge base written in Prolog. In this section, we give a high-level overview of logic, with particular reference to first-order programming logic. Logic programming was invented in 1974 by Kowalski [5]. He proved that first-order logic can be considered a useful and practical programming language with theoretical foundations. This is due to the Horn clauses studied by the logician Alfred Horn (1951) [6]. Horn clauses are the basis of logic programming as they help implement the so-called "linear resolution with selection" function (SL-resolution) [7]. SL-resolution starts

with a query and resolves subsequently with rules and facts until a negation of the query is proved.

Prolog was designed as a programming language and has been extensively used in several research areas, such as molecular biology, design of VLSI (Very Large Scale Integration) systems, legislation, and options trading [8]. It can be used in every domain that can be represented as facts and rules. The language allows us to have a logical representation of the context and to easily generate a program able to solve a problem. Prolog differs from Logic Programming in several aspects. It can be considered a specialization or refinement of a program in Horn Clause Logic, where the selection rule and the search strategy are fixed [9]. A quick introduction to the Prolog syntax can be found in Matuszek's course [10].

3.1 Logic programming and security

Logic programming finds extensive utility within the security domain. Notably, in network security, logic programming serves as a valuable tool for evaluating the security level of communication protocols [11, 12]. Furthermore, Barker and Steve (2000) [13] have adeptly employed logic programming in the formulation of Role-Based Access Control (RBAC) security models. Zech et al. (2013) [14] have demonstrated the potential of logic programming in automating the risk analysis process, thereby enhancing penetration testing endeavors with critical risk insights. Moreover, within the realm of software testing, logic programming emerges as a prevalent approach for generating test cases, as evidenced by numerous studies [15–17].

In another paper published in 2019 [18], Zech et al. expand the realm of test-case generation into the domain of security. The authors showcase the adeptness of a knowledge-based system in uncovering vulnerabilities within web applications. Their approach involves the introduction of a *security problem* concept, which serves to model the System Under Test (SUT) using a Domain Specific Language. The concept of a *security risk profile* is then introduced to identify the most critical vulnerability to which the SUT is susceptible. To achieve this, they devise an expert system encompassing both the security risk model and a grammar-based test data generator. To substantiate their findings, the authors engineer a state machine that can effectively identify SQL injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities. Their innovative formulation of the SUT using Specification and Description Language (SDL) proves intriguing, displaying several points of intersection with our own work. Their framework establishes a correlation between the most salient vulnerability based on security risks and the attack payloads directed at the system. Our approach follows a similar strategy, albeit with notable distinctions. We employ a knowledge-based system to facilitate the oracle module within the fuzzing process. In contrast, the cited authors employ the knowledge base to map threats to the test-data generation procedure. Additionally, our approach takes a semi-automatic route to generate template-based HTTP requests for use in the fuzzing process. This varies from their approach of utilizing a web spider, which can be susceptible to issues like forbidden errors for authenticated pages and missing links generated dynamically before the HTML page rendering phase.

4 Fuzzing: state of the art

Fuzzing, also known as fuzz testing, is an automated software testing technique designed to uncover irregularities in a target application by feeding it with invalid, unexpected, or random input data. Throughout the execution of fuzzing, the target application is closely monitored to unveil software errors that might indicate the presence of a security vulnerability. Although the notion of employing random data to trigger anomalies may appear simplistic, it has yielded impressive outcomes by identifying errors in various software applications.

Over the past two decades, fuzzing has demonstrated its exceptional efficacy in unearthing vulnerabilities that often go unnoticed by static software analysis and manual code reviews. This technique has seamlessly integrated into software development practices and is now considered an indispensable component for assuring software security.

In the literature, fuzzing techniques are categorized into three distinct types: opaque-box, grey-box, and clear-box. The classification hinges on the extent of information available about the System Under Test (SUT) before the fuzzer's execution.

Opaque-box fuzzing techniques operate without access to any internal information about the target system, such as its source code or system documentation. These methods solely rely on input and output data. Often referred to as data-driven or input/output-driven testing, this approach keeps the intricacies of the system hidden. The testing process revolves around observing the outcomes of the system and pinpointing deviations from specified behavior. Complex fuzzers employ generative or mutational input strategies for this purpose [19].

Conversely, clear-box fuzzing techniques differ from opaque-box methods. In a clear-box approach, the tester can tap into supplementary information like source code or design specifications, offering insights into the system's behavior. With this knowledge, it's feasible to enhance the coverage of the system during testing, thereby boosting its overall effectiveness.

A grey-box fuzzer occupies a middle ground, adopting a "lightweight" approach to gather insights about the System Under Test (SUT). It gathers data through statistical analysis, sourced either from a static evaluation of the system

or dynamic data extracted, for example, from a debugger (McNally et al. [20]). This information might not be exact, but it enables faster execution and exploration of input possibilities. Grey-box techniques initiate mutational input exploration with a seed input. When intriguing pathways are activated, the mutated input is scrutinized, prompting further mutations to explore the remaining input spectrum. The objective is to enhance behavioral coverage and unearth vulnerabilities in the system based on information gleaned from the fuzzer [21].

Our work aligns with the opaque-box fuzzing category. It pertains to emulating the actions of a security expert who manually sets up the fuzzing attack. This method becomes essential when access to the source code is unavailable. Consequently, adopting an opaque-box approach offers a viable chance of detecting vulnerabilities without any prior familiarity with the application under examination. This not only enhances the efficiency of vulnerability detection but also diminishes the time needed for a manual security assessment.

4.1 A generic fuzzing algorithm

Valentin J.M. Man et al. (2019) [22] introduced a versatile fuzzing algorithm capable of accommodating the various types of fuzzing methods described above.

Algorithm 1: Generic Algorithm of a fuzzer

Data: $\mathbb{C}$, t_{limit}
Result: $\mathbb{B}$ `// finite set of bugs`
1 $\mathbb{B} \leftarrow \emptyset$
2 $\mathbb{C} \leftarrow preprocess(\mathbb{C})$
3 **while** $t_{elapsed} < t_{limit} \wedge continue(\mathbb{C})$ **do**
4 conf $\leftarrow schedule(\mathbb{C}, t_{elapsed}, t_{limit})$;
5 tcs $\leftarrow inputget$(conf);
`//` O_{bug} `is embedded in a fuzzer`
6 $\mathbb{B}'$, execinfos $\leftarrow inputeval$(conf,tcs, O_{bug});
7 $\mathbb{C} \leftarrow confUpdate$($\mathbb{C}$,conf,execinfos);
8 $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
9 **end**
10 return $\mathbb{B}$

The algorithm accepts a set of fuzz configurations $\mathbb{C}$ and a timeout parameter t_{limit} as input, producing a set of vulnerabilities $\mathbb{B}$ as output. It is structured into two phases: a pre-processing phase, which is executed at the start of a fuzzing campaign, and a subsequent fuzzing process. Notably, some fuzzers may not implement all the functions within the fuzzing process.

The pre-processing step involves working with a set of configurations and results in a modified set of configurations tailored to the requirements of the specific fuzzing algorithm being used. During each iteration, the chosen configuration file is updated using the $schedule$ function, considering both the time that has passed and the preconfigured time limit. This stage tackles a challenge known as Exploration vs Exploitation [23], which revolves around the allocation of computational resources. It entails finding the right equilibrium between exploring new configurations to maximize fuzzer performance and refining the existing configuration to approach the optimal solution.

Subsequently, the scheduling function endeavors to enhance system coverage and optimize performance, potentially altering the set of selected configurations. Following the scheduling phase, the next step is to determine the test input. The $inputgen$ function defines, mutates, or generates a series of test cases to be executed against the program. It assesses whether these executions result in security policy violations. Vulnerability detection is achieved by consulting an oracle. Finally, the configuration parameters are adjusted based on the discovered vulnerabilities, using the information acquired during the current iteration through the $confUpdate$ function. The algorithm concludes once the specified number of iterations is completed.

4.2 Related works and fuzzing problems

One of the challenges in fuzzing is selecting the configuration that best optimizes performance. Fuzzer designers must analyze the available information and make a choice that leads to an ideal outcome, such as discovering the maximum number of vulnerabilities in the shortest possible time. This challenge is often framed as the Exploration vs Exploitation problem, where the decision revolves around whether to explore new configurations or maximize the performance of a chosen configuration. Woo et al. (2013) [24] referred to this challenge as *Fuzz Configuration Scheduling*. Berry and Fristedt (1985) [25] delved into the problem of searching for the optimal configuration. They allocated a finite set of resources to various configurations and aimed to maximize the achievable gain. They found that a configuration minimizing the consumption of temporal resources tended to discover a greater number of vulnerabilities. Householder et al. (2012) [26] took advantage of information from a mutational black-box fuzzing execution to define an ideal configuration that maximized vulnerability discovery. They achieved this by modifying the CERT Basic Fuzzing Framework (BFF) algorithm, resulting in an 85% increase in crashes.

Discovering vulnerabilities in a web application requires a fuzzer to effectively differentiate between expected (normal) behavior and unexpected (buggy) behavior. However, distinguishing a bug from a feature can be challenging, leading to what is known as the *Oracle Test Problem* [27]. An oracle test is employed to determine whether a test is successful or not, involving a comparison of the system's outputs, given a specific input, with the expected outputs during the application's normal execution.

Table 1 Inspirational contributions from relevant works focused on software testing and their impact on the design of our solution

Work	Contribution	Impact on authors' work
Berry and Fristedt (1985) [25]	Search for the optimal configuration to maximize performance by reducing resource consumption	We explore payload set optimization strategies and propose several optimization criteria
Householder et al. (2012) [26]	Adopt a mutational black-box fuzzing approach to maximize vulnerability discovery	We maximize vulnerability discovery through optimization criteria based on security expertise. Our fuzzer can nonetheless be easily adapted with other algorithms, such as mutational ones
Woo et al. (2013) [24]	Define the "Fuzz Configuration Scheduling" task as a challenge to maximize fuzzer's performance	We define an initialization phase to configure relevant settings, such as the proxy, payloads, and observations. The fuzzer can be adapted to maximize its performance
Barr et al. (2015) [27]	Provide a detailed survey of the oracle problem in software testing	We define the vulnerability discovery phase as an oracle problem, wherein we design a specific module to interact with a knowledge base. This module detects vulnerabilities by analyzing anomalies stimulated during the fuzzing phase

Although these prior studies were focused on defining fuzzing solutions for the software testing domain, they significantly influenced the design of our rule-based fuzzer. We incorporate an initial configuration exploration strategy and a detection approach that combines an Oracle test with an expert system. The Oracle test is designed to identify discrepancies between valid interactions with the application and those of an attacker. These differences serve as observations used to query a vulnerability knowledge base containing a set of rules derived from existing security expertise. This knowledge base is constructed by inferring an application's behavior in the context of injection-type vulnerabilities. Table 1 summarizes the contributions of the related works mentioned above, which have significantly influenced the design of our solution.

Other works deepen the research of fuzzing solutions to discover vulnerabilities in web applications. Duchene et al. (2014) [28] used a black-box fuzzer to detect Cross-Site Scripting (XSS) vulnerabilities with the assistance of an Oracle test. Their approach utilized a genetic algorithm to generate fuzz inputs and then compared the Document Object Model (DOM) obtained after a cross-site scripting injection with established behavioral patterns. They represented the DOM injection as a taint tree. Similarly, Appelt et al. (2014) [29] adopted a comparable approach to identify SQL injection vulnerabilities. They employed a mutational approach that intercepted communications between the web application and the underlying database to uncover these vulnerabilities.

Khalid et al. (2018) [30] employed a predictive white-box fuzzer approach to detect vulnerabilities. In their work, the authors analyzed the input validation and sanitization processes within an application to predict the presence of vulnerabilities. This predictive method combines insights from both static and dynamic code analysis, enhancing the accuracy of vulnerability prediction.

Current research endeavors are focused on adapting artificial intelligence models to grey-box fuzzing for the detection of cross-site scripting vulnerabilities [31, 32]. However, these approaches are often tailored to specific programming languages and designed to identify a single type of vulnerability. Additionally, the grey-box approach can be challenging to implement, especially when the source code of the assessed application is not available.

When selecting a web application scanner, it is critical to thoroughly assess its features. Each scanner has its unique characteristics and capabilities. Altulaihan et al. (2023) [33] have delved into this area, offering recommendations for the best scanners and proposing tool combinations to achieve optimal performance. We have drawn inspiration from their guidance in developing our fuzzer, which capitalizes on the knowledge provided by security experts to enhance its capa-

bilities. As stated in Sect. 8, there does not exist a formalized and comprehensive test suite useful to obtain an objective performance evaluation. Furthermore, the related works do not publicly provide the source code, making a comparison challenging. However, it is helpful to summarize the strengths and limitations of each approach compared to ours. Tables 2 and 3 show such a comparison. Related works typically integrate white-box or grey-box approaches with innovative algorithms or AI-based techniques to enhance the discovery of specific classes of vulnerabilities. However, they often struggle to cover various programming languages or vulnerabilities comprehensively. In future endeavors, we aspire to develop a formalized benchmark to replicate experiments and facilitate a quantitative comparison with related works.

As evident from the tables, each work employs different benchmarks and performance metrics. Hence, in Sect. 6, we endeavor to compare our approach with a prominent open-source scanner. Moving forward, we plan to address this challenge by developing a comprehensive benchmark, thereby enhancing the evaluation process. Section 8 delves into this issue, offering suggestions to refine the proposed work.

In summary, our work employs a black-box, rule-based methodology to detect input-handling vulnerabilities. We have devised specific vulnerability rules targeting SQL injection, reflected XSS, and Path Traversal. The vulnerability discovery process entails validating the integrity of data retrieved from the application and subsequently consulting the expert system to identify any breaches of security policies.

5 A rule-based fuzzer to discover input handling vulnerabilities

The methodology described in Sect. 2 is designed to uncover vulnerabilities in various areas, including authentication, client-side functionality, and input handling. As demonstrated in [35], fuzzing can effectively identify *input handling flaws* in web applications. The process of discovering input-handling vulnerabilities involves a series of sequential steps. Initially, the tester identifies the entry points of the application. Then, they inject malicious payloads to provoke abnormal behaviors. Finally, they conduct further investigation to confirm the presence of a vulnerability. Fuzzing is a valuable technique for generating anomalies that can reveal the existence of flaws and vulnerabilities. However, selecting appropriate input for fuzzing is a complex task, as the input must be semantically correct to avoid rejection by the application. In the context of web applications, input primarily consists of HTTP requests. Nevertheless, a single web application can expose multiple input pages, and each HTML page may contain numerous parameters that can be subjected to fuzzing. Security testers often manually instrument the input by interacting with the tested web application and developing tools that replicate these interactions. In our platform, we have two modules that address this challenge.

The *Interceptor Module* is responsible for capturing and storing web interactions that occur during the initial interaction between the security tester and the target website. This functionality enables the system to gain insights into how to effectively engage with the target. Meanwhile, the *Repeater Module* replaces the values of request parameters with placeholders and generates a generic *repeater file* that can be employed in the fuzzing phase. In the realm of web application penetration testing, testers often lack comprehensive knowledge of the inner workings of the target system. Consequently, opaque-box fuzzing becomes the most suitable technique. Following the methodology for testing web applications, testers devise malicious payloads aimed at uncovering various types of vulnerabilities. For instance, when seeking to identify an SQL Injection vulnerability, testers create a list of payloads that have the potential to trigger SQL errors. They then initiate the fuzzing process using these payloads and examine the outcomes. If errors are encountered, anomalies are scrutinized to validate the presence of a vulnerability. This final step can be systematically modeled by employing a rule-based system that incorporates the criteria established by security experts to confirm the existence of vulnerabilities.

As previously mentioned, our primary focus revolves around three distinct types of injection vulnerabilities: Cross-Site Scripting, SQL Injection, and Path Traversal. Nevertheless, it is important to note that our designed solution boasts a high degree of customization and can be readily extended to detect other categories of vulnerabilities. The vulnerability discovery phase relies on expert knowledge that has been formally represented in the form of a knowledge base. This approach enhances the efficiency of detection. The knowledge base contains information concerning the correlations between anomalous responses from the target and potential vulnerabilities, as discerned by experienced security testers. Essentially, it operates on the same principles as an actual security expert who launches attacks against a target and assesses for vulnerabilities when anomalous behaviors stemming from errors manifest.

In the following sections, we will delve into the algorithm, the system architecture, and the interactions that transpire throughout the vulnerability discovery process.

5.1 Initialization and fuzz-run phases

The Rule-Based fuzzer algorithm takes a set of configuration files $\mathbb{C}$ as input:

Table 2 Benchmarks and performance metrics used in related works compared with our approach

Work	Used benchmark	Number of tests	Perf.
Duchene et al. (2014) [28]	Three intentionally vulnerable web applications and four known web applications	32 (XSS)	100% TP 0% FN
Appelt et al. (2014) [29]	SugarCRM and HotelRS	33 ops, 108 params	15,4% exploited SQLi
Khalid et al. (2018) [30]	Drupal, PhpMyAdmin, and Moodle	113 (SQLi)	Recall: 0,46 Precision: 0,53
Liu et al. (2023) [31]	Karroum, WhiteRabbic, and XSSed	10,120 (XSS)	Recall: 0,997
Song et al. (2023) [32]	WebGoat and Jeens−1.4.2	49 (XSS)	100% TP 0% FN
Our approach	Wavsep	297 (SQLi, PT and XSS)	See Tables 7 and 18

Table 3 Qualitative comparison with related works focused on web applications

Work	Strengths	Limitations
Duchene et al. (2014) [28]	Headless-browser to reduce false positives	Only cross-site scripting vulnerabilities (though with suggestions for potential extensions)
Appelt et al. (2014) [29]	Interesting evaluation approach that includes WAF (Web Application Firewall) protection	Only SQL injection vulnerabilities; no performance metrics such as accuracy and precision
Khalid et al. (2018) [30]	Good vulnerability dataset for training the software [34]; clear evaluation and comparison, achieved by reproducing experiments conducted in other existing studies	Only SQL injection vulnerabilities; only PHP-based web applications
Liu et al. (2023) [31]	A complete dataset that also includes real-world vulnerabilities; generic approach based on JavaScript snippets, capable of covering all types of server-side languages	Only cross-site scripting vulnerabilities based on JavaScript
Song et al. (2023) [32]	Use of a headless-browser to reduce false positives	Only cross-site scripting vulnerabilities; only Java-based web applications; the evaluation can be improved by using a larger dataset
Our approach	Use of a headless-browser to reduce false positives; covers multiple vulnerabilities; covers multiple programming languages; integration of security expertise optimizes performance	The rule-based approach can be improved through AI-based techniques (see Sect. 8); the evaluation can be improved by designing a novel web application vulnerability benchmark

- *Proxy configuration*: a configuration file containing the information needed to setup the MITM (Man In The Middle) proxy module, used to capture HTTP interactions between the user and the target web application;
- *Payload configuration*: a configuration file containing information related to the attack strings used during fuzzing to trigger errors in the target web application. In our tests, we define SQLi, PT, and XSS payloads;
- *Observations configuration:* a configuration file that specifies the types of observations to be processed by the vulnerability knowledge base. In subsequent sections, we will provide a detailed explanation of these observations when we describe the functionality of the analyzer module in our platform;
- *Relevant strings configuration*: a configuration file containing keywords that will be searched for inside the content of an HTTP response. We will explain the role of relevant strings further in the paper.

When the algorithm execution is completed, the system outputs a set of vulnerabilities $\mathbb{B}$.

The *preprocess()* function processes the information contained in the configuration files to define the execution environment. During this phase, the system is instrumented

Algorithm 2: Rule-Based Fuzzer Algorithm

```
Data: Input: ℂ; Output: 𝔹 // finite set of bugs
1  𝔹 ← ∅
2  ℂ ← preprocess(ℂ)
3  conf ← schedule(ℂ);
4  while continue(ℂ) do
5  |  tcs ← inputgen(conf);
   |  // O_bug is embedded in a fuzzer
6  |  execinfos ← intruder(conf,tcs);
7  |  analyzeinfo ← analyzer(conf,execinfos);
8  |  𝔹′ ← eval(analyzeinfo, O_bug);
9  |  𝔹 ← 𝔹 ∪ 𝔹′
10 end
11 return 𝔹
```

by configuring the proxy parameters, specifying the payloads, and defining relevant strings. The *schedule*() function defines the configuration to be used within the fuzz-run phase. In our approach, the mentioned *schedule* function does not make any optimal configuration choice in the configuration space. It rather uses the same configuration as the one defined in the preprocessing phase, essentially acting as an identity function.

The intrusion and vulnerability detection phases follow the configuration phase. The *inputgen*() function generates test cases using the previously defined payloads. In the context of web applications, these test cases are HTTP requests with malicious payloads inserted at specific points within the request. The intrusion phase occurs during the execution of the *intruder*() function, which processes the test cases and attacks the target application using a "Sniper attack", as will be explained in more detail later in the paper. As a result of the intruder phase, an output file containing information about the results of the fuzzing attack is generated.

The detection phase involves evaluating the results to identify vulnerabilities using the *analyzer*() and *eval*() functions. The *analyzer* function compares the observations obtained from a valid HTTP interaction with those generated during the fuzzing of the web application. An oracle module then checks these observations based on its knowledge base. The Oracle test is a critical component in this phase as it verifies the existence of specific conditions that indicate the presence of a vulnerability. The oracle checks for vulnerabilities by querying a knowledge base containing a set of rules and assertions that formalize anomalous observations. The *eval()* function retrieves information from the Oracle tests and determines whether a security policy is being violated. A policy violation occurs when the system concludes that the observed anomalies are indeed indicative of a vulnerability.

Figure 1 presents a high-level overview of the Rule-Based Fuzzer. As shown, the system consists of several independent modules. This modular design enables security testers to execute each fuzzing phase individually. Additionally, we employed the system to configure the files using the process outlined in Algorithm 3. Another advantage of this modular approach is the ability to repeat experiments consistently.

For instance, it is possible to modify configuration files, optimize the payload set, enhance the knowledge base, and perform the fuzzing step without having to redo the previous ones.

5.2 Proxy

The Proxy module intercepts and records HTTP interactions generated during the user's navigation. This acquisition is accomplished using a man-in-the-middle proxy positioned between the client and the server. As illustrated in Fig. 2, we have implemented this proxy using Mitmproxy [36], an interactive proxy that supports SSL/TLS for HTTP/1, HTTP/2, and WebSocket. The module is configured as a transparent proxy (see Fig. 3), also known as an intercepting proxy, inline proxy, or forced proxy. In this mode, it intercepts communications at the network layer (layer 3 of the ISO/OSI stack) without requiring any client-side configuration [37]. This approach allows us to intercept HTTP session requests without modifying them. The creation of a custom module that incorporates transparent proxy functionality was necessary to select the injection points of interest in an HTTP request. Listing 1 presents a simple HTTP POST request, which serves as a typical example of input for one additional component of our architecture, namely the *Repeater* module.

```
POST /search.php?test=query HTTP/1.1
Host: testphp.vulnweb.com
Connection: keep-alive
Content-Length: 31
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://testphp.vulnweb.com
Content-Type: application/x-www-form-
              urlencoded
User-Agent: Mozilla/5.0
(Windows NT 10.0; Win64;...
Referer: http://testphp.vulnweb.com/search.
          php?
test=query
Accept-Encoding: gzip, deflate
Accept-Language: it-IT,it;q=0.9,
en-US;q=0.8,en;q=0.7

searchFor=injection&goButton=go
```

Listing 1: Post Request Example

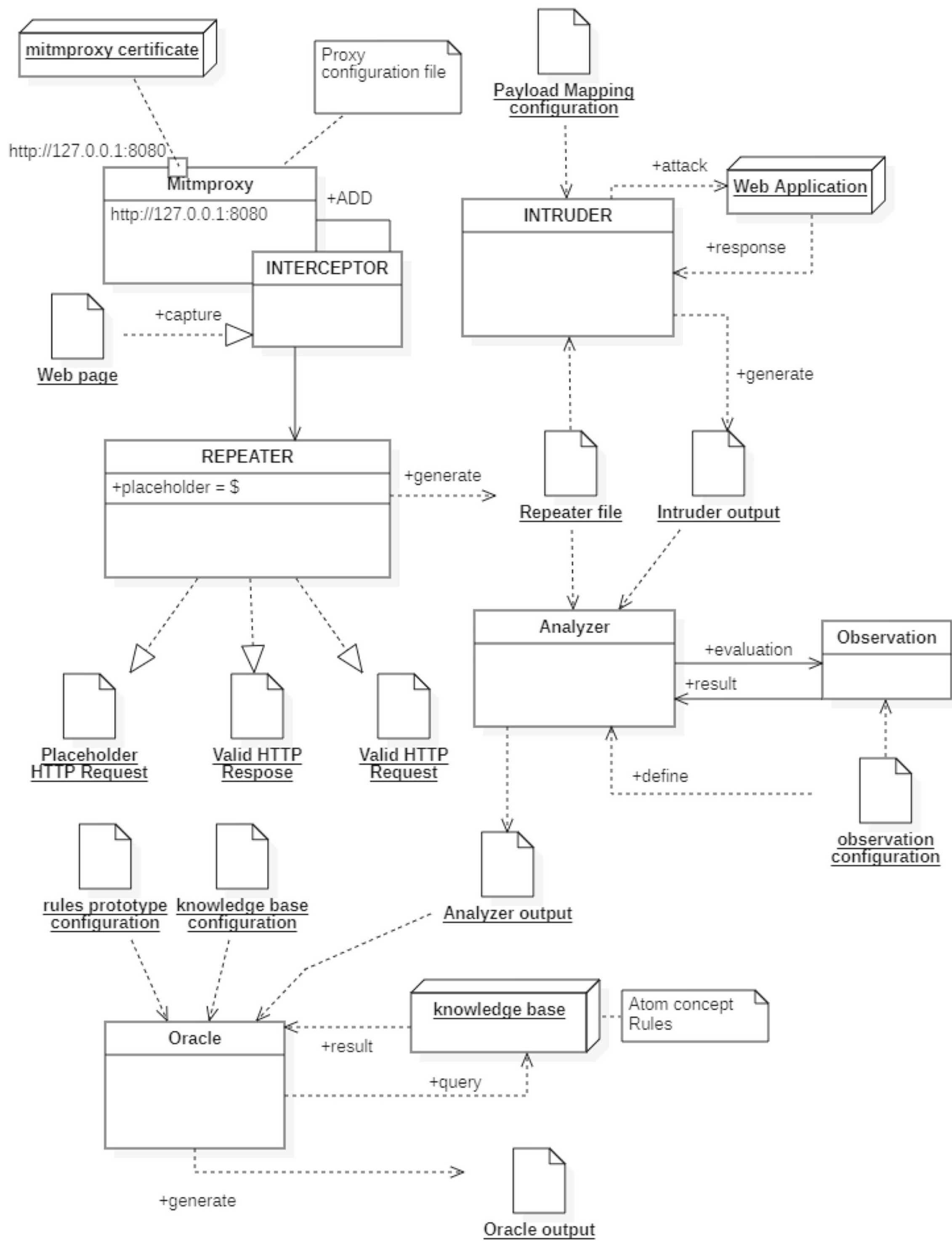

Fig. 1 Rule-Based Fuzzer - High-Level Architecture

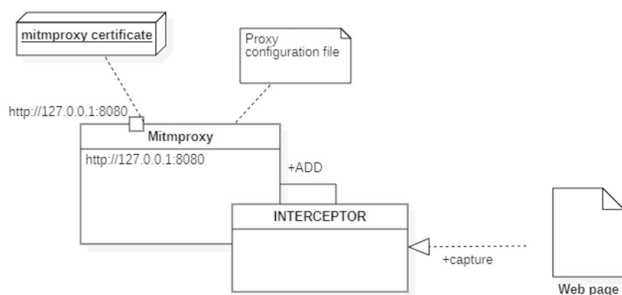

Fig. 2 High-Level Architecture - Mitmproxy Interceptor

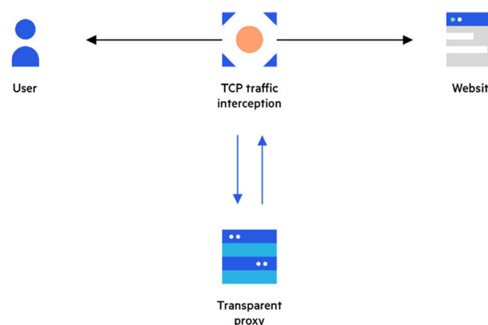

Fig. 3 Transparent Proxy

5.3 Repeater

Because the system operates as an opaque-box fuzzer, it cannot automatically deduce the schema of executed requests. The repeater module enhances Mitmproxy's capabilities by inspecting the sent HTTP requests and substituting placeholders for injection points. An injection point might represent any pertinent attribute of an HTTP request, such as a parameter or a header value. The outcome of the interceptor module is a collection of HTTP requests with placeholders.

Definition 1 (Placeholder HTTP request) Let:

- $\mathbb{HR}$ the set of HTTP requests intercepted by the MITM proxy component;
- $\mathbb{IP}$ a set of injection points of $hr \in \mathbb{HR}$ defined in the Proxy configuration file;
- placeholder symbol a fixed-size string, like, e.g., `$placeholder$`.

A Placeholder HTTP Request is the output of a function that replaces all of the injection points with the placeholder symbol:

$$SetupPlaceholder(hr, \mathbb{IP}), hr \in \mathbb{HR}$$

As highlighted in Fig. 4, the Repeater module outputs a set of data in the following form:

- **idFuzz:** session fuzzing identifier;
- **Request:** a valid HTTP request;
- **Response:** a valid HTTP response;
- **Placeholder HTTP request:** a valid HTTP request containing placeholders at selected parameter places.

An example of how the HTTP request in Listing 1 might look after having been processed by the Repeater is reported in Listing 2.

The output of the repeater module will be used in the fuzzing phase.


```
"id_fuzz_1": {
   "Request": {
   "method": "POST",
   "url": "http://sitevuln.org/search.php?
   test=query",
  "header": {"User-Agent": "Mozilla/5.0
  (Win...,},
  "payload": "searchFor=test&goButton=go"
    },
    "Response": {
  "url": "http://sitevuln.org/search.php?
  test=query",
  "status_code": 200,
  "header": {"Server": "nginx/1.19.0",...},
  "time_elapsed": "0:00:00.031389",
  "content_length": 4773,
  "html": "<!DOCTYPE HTML PUBLIC
 \"-//W3C//DTD ..."
    },
  "PlaceholderRequest": {
  "method": "POST",
  "url": "http://sitevuln.org/search.php?
  test=$query$
  "header": {"User-Agent": "Mozilla/5.0... },
  "payload": "searchFor=$test$&goButton=$go$"
    }
}
```


Listing 2: JSON data before the fuzzing phase

5.4 Intruder

The Intruder module, whose high-level architecture is described in Fig. 5, performs a fuzzing against the target application with several attack payloads that specialize the placeholder HTTP requests generated by the Repeater module. The fuzzing session produces a set of HTTP interactions, together with additional information (see Listing 3):

- **idFuzz**: session fuzzing identifier;
- **Request**: the HTTP request used for fuzzing;
- **Response**: the related HTTP response;
- **TypePayload**: the type of payload used during the fuzzing session, e.g., SQLi, XSS, and PT;

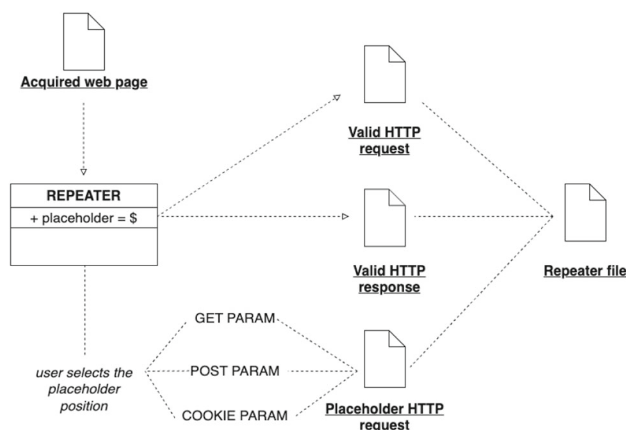

Fig. 4 High-Level Architecture - Repeater

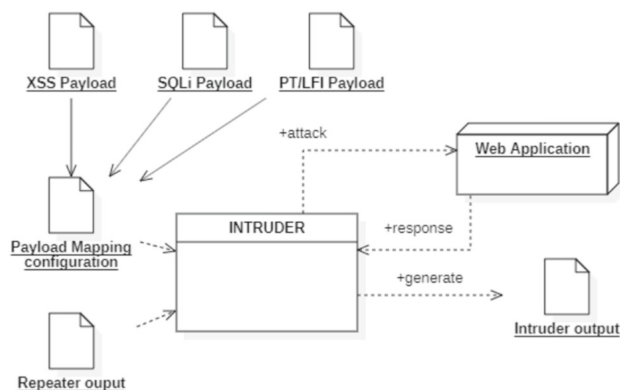

Fig. 5 High-Level Architecture - Intruder

- **Payload** the payload string used during the fuzzing session.

The attack on the system is carried out using one or more payload lists that trigger the aforementioned vulnerabilities. Payloads were selected from several public lists, as well as by deeply investigating the behavior of several web application scanners. Collected payloads were fine-tuned during the vulnerability knowledge base generation process defined in Algoritm 3. We decided to retain:

- **59** PT payloads;
- **67** SQLi payloads;
- **15** XSS payloads.

In Sect. 6, we show that the accuracy depends on both the type and the number of payloads used during the fuzzing process. In this work, we use an empirical approach to choose the payloads by simply collecting those that generate the most interesting analyzer observations. Payload set optimization is discussed in more detail in Sect. 7. To fuzz the application, the *Sniper Attack* is used.


```
"id_fuzz_1": {
 "Results": [{
  "Request": {
   "method": "POST",
   "url": ".../search.php?test= '
  UNION SELECT 1...",
   "header": {"User-Agent": "Mozilla/5.0
  (Wind,..."},
   "payload": "searchFor=aaaa\u00a7goButton=go"
   },
  "Response": {
   "url": ".../search.php?test=
   UNION
   "status_code": 200,
   "header": {"Server": "nginx/1.19.0..."},
   "time_elapsed": "0:00:00.035058",
   "content_length": 6720,
   "html": "<!DOCTYPE HTML PUBLIC
\"-//W3C//DTD..."
   },
   "TypePayload": "sqli",
   "Payload": " ' UNION SELECT 1, table_name,
 'jfks'..."},

{"Request": {"..."},
"Response": {"..."},
"TypePayload": "sqli",
"Payload": " 1 UNION SELECT 1..."}
...
}
```


Listing 3: JSON data obtained after the fuzzing phase

Definition 2 (Sniper Attack) Let *phr* a placeholder HTTP request composed of n injection points ip, and $\mathbb{FS}$ a set of m fuzzing strings defined in the payload configuration file.

For each ip, generate m requests by replacing the ip placeholder with all m fuzzing strings in $\mathbb{FS}$ while configuring the other injection points with valid values. The sniper attack is performed by sending to the target web application the $n \times m$ HTTP requests obtained in such a way.

This type of attack is beneficial when you want to test several request parameters individually for the same type of vulnerability. It is also useful to evaluate efficiency, as it is possible to compute the number of performed HTTP requests by multiplying the number of placeholders by the number of used payloads.

5.5 Analyzer

The high-level structure of the Analyzer module is reported in Fig. 6. The aim of such a component is to convert the collected fuzzing interactions into information understandable by the Oracle knowledge base. The hearth of the vulnerability discovery phase is indeed the *observations' analysis*. To confirm a vulnerability, a security expert typically observes the differences between a valid HTTP

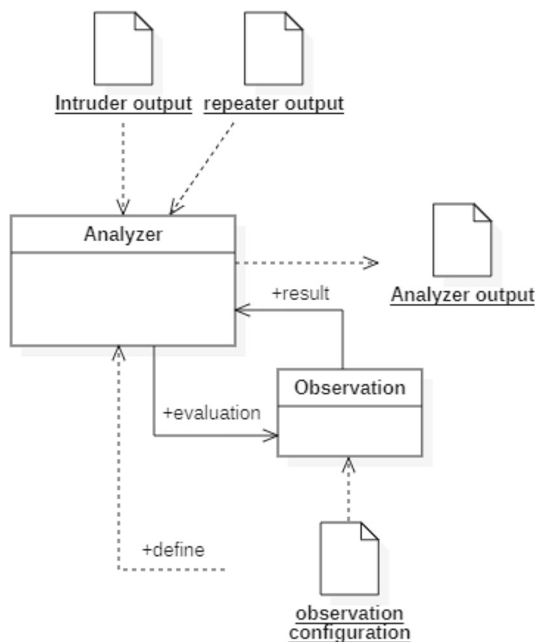

Fig. 6 High-Level Architecture - Analyzer

response and the collected fuzzing responses. We replicate the same approach by comparing the performed HTTP interactions. As said, the Repeater module extracts the result of a valid HTTP interaction, whereas the Intruder module collects the information generated by the fuzzing phase.

After collecting the HTTP responses, the Intruder parses them to extract relevant information and check for the presence of potential vulnerabilities. We call such information *HTTP interaction features*.

Definition 3 (HTTP interaction features) Let:

- $\mathbb{HI}$ the set of HTTP interactions obtained through a fuzzing attack;
- $i \in \mathbb{HI}$ a single HTTP interaction;
- $p(i), i \in \mathbb{HI}$ a parser function of the intruder module.

We define `HTTP interaction features` as the set of values returned by $p(i)$.

We identify the following useful HTTP interaction features:

- **Payload Type**: the categorized payload, namely SQLi, XSS, PT, or Normal (for valid HTTP interactions;
- **Payload String**: the payload string utilized during the fuzzing attack;
- **HTTP status code**: the HTTP response code associated with the sent HTTP request;
- **Content-Length**: length in bytes of the obtained HTTP response;
- **Time-elapsed**: elapsed time in microseconds from the time the HTTP request is sent to the time the associated response arrives;
- **Body response**: the content of the HTTP response.

Payloads are categorized by the searched vulnerability. Categorization is performed to decrease the number of false positives. Such an approach mimics the methodology adopted by security experts to exclude false positives during their tests. After fuzzing the web application, obtaining the responses, and analyzing a potential vulnerability, they verify if the used payloads may trigger it. During the fuzzing attack, we collect the payload types. The oracle will select a subset of rules from the knowledge base depending on the used payload. The Analyzer compares valid and fuzzing feature interactions and generates what we define *Analyzer Observations*.

Definition 4 (Analyzer Observations) Let:

- $\mathbb{HI}$ the set of HTTP interactions obtained through a fuzzing attack;
- $i_{fuzz} \in \mathbb{HI}$ a single HTTP interaction of the fuzzing attack;
- i_{valid} a valid HTTP interaction;
- f_{valid} the HTTP interaction features of i_{valid};
- fi_{fuzz} the HTTP interaction features of i_{fuzz}.

Then, Analyzer Observations is a set of observations returned by a function of f_{valid} and fi_{fuzz}: $analyze(f_{valid}, fi_{fuzz})$

The analyzer observations are the relevant observations commonly investigated by a security expert to verify the presence of a vulnerability. They are generated by comparing the HTTP interaction features of a valid HTTP interaction with those captured during a fuzzing attack.

During a security testing activity, security experts look for several strings within the response to confirm the presence of a vulnerability. For example, when performing an SQLi attack, they might be checking for the existence of string sequences like the following one: "`You have an error in your SQL syntax`".

Another relevant analysis is about content length and elapsed time anomalies, which typically unveil the presence of blind-based vulnerabilities.

Based on the above considerations, the analyzer observations are essentially a formalized representation of the security expert's analysis:

- **Relevant strings**: tokens of interest extracted from the body of fi_{fuzz};
- **Payload string**: the payload field of fi_{fuzz};
- **Payload Type**: the payload type field of fi_{fuzz};

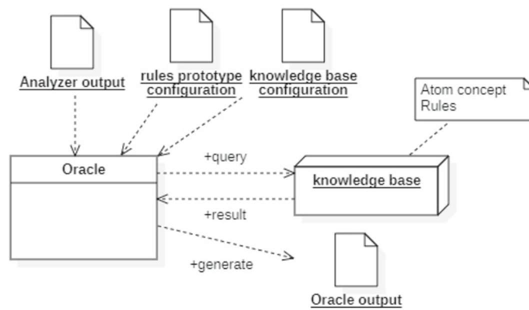

Fig. 7 High-Level Architecture - Oracle

- **HTTP status code**: the return code field of fi_{fuzz};
- **Anomalous Content-Length**: a boolean value configured based on the difference between the content-length field values associated, respectively, with f_{valid} and fi_{fuzz}. If such a difference is above a predefined threshold value, *Anmalous Content-Length* is set to true, false otherwise;
- **Anomalous response time delay** a boolean value configured based on the difference between the elapsed time field values associated, respectively, with f_{valid} and fi_{fuzz}. If the difference is above a predefined threshold value, *Anomalous response time delay* is set to true, false otherwise.

5.6 Oracle

The vulnerability discovery phase is accomplished through the Oracle module, which interacts with a *vulnerability knowledge base* to verify if the detected anomalies can be ascribed to the presence of vulnerabilities.

Hence, the vulnerability knowledge base leverages the logic programming declarative semantics to define a relationship between analyzer observations and vulnerabilities. The knowledge base is composed of `terms`, `predicates`, and `rules` categorized by the vulnerability type. Figure 7 shows the Oracle architecture. The Oracle module receives the analyzer observations and, based on the rules contained in the knowledge base, determines whether vulnerabilities exist in the tested web application. When considering the payload type, a specific subset of rules is activated. This means that the knowledge base rules are only triggered if they pertain to a particular vulnerability, as verified by checking the payload type used during the attack. To formalize these considerations, we introduce the concept of the vulnerability knowledge base.

Definition 5 (Vulnerability Knowledge Base)
Let:

- $\mathbb{T}$ the finite set of vulnerability types (SQL injection, XSS, etc.);
- $t \in \mathbb{T}$ a vulnerability type; If $t, q \in \mathbb{T}$, then $t \neq q$.
- $\mathbb{P} = \mathbb{T}$ the finite set of payload types;
- $\mathbb{AO}$ a set of analyzer observations obtained from the execution of the $analyze()$ function, with:
  - s the status code of $\mathbb{AO}$;
  - r_s the relevant strings of $\mathbb{AO}$;
  - $pt \in \mathbb{P}$ the payload type of $\mathbb{AO}$;
  - $a_{content_length}$ the anomalous content-length value of $\mathbb{AO}$;
  - $a_{response_time}$ the anomalous response time delay of $\mathbb{AO}$.

Then:

- analyzer observations can be modeled as *Prolog arguments*. We call them **vulnerability observations**. We define $\mathbb{VO}$ as the set of vulnerability observations;
- s, r_s, pt, $a_{content_length}$, $a_{response_time}$ values can be modeled as *Prolog facts*;
- $\mathbb{VKB}$ is a set of vr vulnerability rules;
- $\mathbb{VKB}_t$ is a subset of $\mathbb{VKB}$ containing all the vr_t vulnerability rules for a given vulnerability type t;
- $\mathbb{VKB}_t$ forms a $\mathbb{VKB}$ partition[2] ;
- A vulnerability rule vr is a conjunction of a subset of vulnerability observations:

  $$vr = vo_1 \wedge vo_2 ... \wedge vo_m$$

  where $0 \leq m \leq \mathbf{card}(AO)$; $vo_i \in \mathbb{VO}$;
- The application is vulnerable to t when a $vr_t \in \mathbb{VKB}_t$ is true;
- An application is vulnerable to v when at least one vulnerability rule is true. This property can be modeled as a disjunction of vulnerability rules:

  $$is_vulnerable = vr_1 \vee vr_2 ... \vee vr_m$$

Oracle output (for which a sample excerpt is reported in Listing 4) is composed of the following fields:

- **idFuzz:** session's fuzzing identifier;
- **Request:** the HTTP request used for fuzzing;
- **Response:** the HTTP response generated by the fuzzing attack;
- **TypePayload:** the payload used during the fuzzing attack;
- **Payload:** the specific payload string tested during the fuzzing attack;
- **Observation:** an observation reported by the Analyzer;

[2] https://en.wikipedia.org/wiki/Partition_of_a_set.

- **Oracle:** the Oracle response in terms of vulnerability rules fired as true.


```
"id_fuzz_1": {
 "Results": [
 {
  "Request": {..."method": "POST",... },
  "Response": {...,
  "time_elapsed": "0:00:00.035058",
  "content_length": 6720...},
  "TypePayload": "sqli",
  "Payload": " ' UNION SELECT 1 ...",
  "Observation": {
  "StatusCode": 200,
  "SearchKeyword": {
    "File not found": 0,
    "No such file": 0,
    "uid=": 0
    ...
  },
  "TimeDelay": 0,
  "ContentLength": 0
  },
  "Oracle": [{
  "rule": "ruleSQLI3(
  "value": {
    "StatusCode": 200,
    "ContentLength": 0,
    "TimeDelay": 0,
    "SearchKeyword": {
        "SQL": 1
  }
},
  "result": true}]
  },
}
```


Listing 4: JSON data generated by the Oracle

As mentioned previously, we have developed a process called the *vulnerability knowledge base generation process* to construct the vulnerability knowledge base. This involved analyzing abnormal behaviors observed during penetration tests conducted in academic labs that focused on input-handling vulnerabilities. In particular, we solved various PortSwigger labs,[3] which are known for their emphasis on input-handling vulnerabilities. Through this process, we continuously refined our knowledge base. Additionally, during our training sessions, we further enhanced the attack payloads by fine-tuning them. Algorithm 3 shows the vulnerability knowledge base generation process.

We used the *Sniper Attack* with a predefined set of attack payloads. Failure in solving the lab might occur for two reasons:

Algorithm 3: Vulnerability knowledge base generation process

Input : Port Swigger labs (a set of payloads)
Output: The vulnerability knowledge base (an improved set of payloads)

(1) **foreach** *input_handling_lab in Port Swigger labs* **do**
(2)   *valid_resp* ← `HttpRequest`(*req*);
(3)   *valid_obs* ← `Analyzer`(*valid_resp*);
(4)   *responses* ← `SendSniperAttack`(*payloads*);
(5)   *observations* ← `Analyzer`(*responses*);
(6)   **if** `VulnNotTriggered`(*observations, valid_obs*) **then**
(7)   *rules* ← `RulesTuning`(*observations, valid_obs*);
(8)   *payloads* ← `PayloadTuning`(*payloads, rules*);
(9) *knowledge_base* ← `RuleMinimization`(*rules*);
(10) *payloads* ← `PayloadMinimization`(*payloads*);
(11) **return** *(knowledge_base, payloads)*;

- the absence of a payload capable of triggering the vulnerability;
- the absence of relevant analyzer observations that reveal the vulnerability.

Hence, after a lab failure, we repeated it by tuning both payloads and analyzer observations. At the end of the process, we performed a further optimization step to reduce the number of payloads and knowledge base rules as much as possible. We basically applied the well-known logic minimization [38] process to obtain the minimal number of rules.

Also, the payloads were fine-tuned by selecting the minimal set that triggers the vulnerabilities contained within the chosen labs.

6 Evaluation

In this section, we will present the performance metrics that we have defined to evaluate the effectiveness of our rule-based fuzzer. These metrics are used to assess the performance of our approach on a benchmark consisting of both vulnerable and non-vulnerable test cases. We will discuss the limitations and strengths of our approach and compare its performance with a well-known injection vulnerability scanner commonly used in the scientific community to detect vulnerabilities in web applications.

6.1 Benchmark and performance metrics

Our rule-based fuzzer was tested on a benchmark containing a set of test cases derived from the well-known WAVSEP (Web Application Vulnerability Scanner Evaluation Project) benchmark [39]. WAVSEP is a web application vulnerability benchmark designed to help assess the performance of web

[3] https://portswigger.net/.

vulnerability scanners. We selected several input-handling test cases from the WAVSEP benchmark, namely:

- 125 test cases vulnerable to SQLi;
- 117 test cases vulnerable to PT;
- 55 test cases vulnerable to XSS.

To evaluate the fuzzer performance in the absence of vulnerabilities, we added additional test cases to the WAVSEP benchmark. In particular, additional 103 non-vulnerable test cases were added. To this purpose, we selected the latest version (version 5.6 at the time of writing) of WordPress, which is a widely used Content Management System for building web applications. We designated certain test cases as non-vulnerable because there were no known input-handling vulnerabilities identified for the specific version of WordPress used in our testing sessions.

Fuzz results are classified into:

- **True Positives — TP:** vulnerable test cases that the rule-based fuzzer properly detects;
- **True Negatives — TN:** non-vulnerable test cases that the rule-based fuzzer rightfully ignores;
- **False Positives — FP:** non-vulnerable test cases that the rule-based fuzzer erroneously classifies as vulnerable;
- **False Negatives — FN:** vulnerable test cases that the rule-based fuzzer erroneously classifies as non-vulnerable.

We define the following performance metrics:

- **Accuracy** $= \frac{TP+TN}{TP+TN+FP+FN}$
- **Precision** $= \frac{TP}{TP+FP}$
- **Recall** $= \frac{TP}{TP+FN}$

Accuracy, precision, and recall can be defined as follows:

- **Accuracy**: is the proportion of the total number of vulnerable and non-vulnerable test cases to the total number of test cases;
- **Precision**: is the proportion of vulnerable test cases correctly detected as vulnerable to the total number of test cases detected as vulnerable;
- **Recall**: is the proportion of vulnerable test cases correctly detected as vulnerable to the total number of vulnerable test cases.

Accuracy is the most intuitive performance metric and is simply a ratio of correctly detected test cases to the total number of test cases. A good performance is not a direct consequence of high accuracy. The metric becomes relevant with a uniform set of false positive and false negative tests. High precision refers to a low false-positive rate. Finally, recall, also referred to as *true rate*, defines how many vulnerable test cases were labeled as vulnerable among all the vulnerable cases. A high recall value implies a high vulnerability detection capability.

In addition to the performance metrics mentioned earlier, we have also defined a few additional support metrics to provide a more comprehensive evaluation of the rule-based fuzzer's behavior. One such metric is *efficiency*, which measures the number of HTTP requests required to complete a fuzzing campaign. This metric helps us assess how efficiently the fuzzer performs in finding vulnerabilities. Another metric we use is the *Number of Fuzzing Payloads* (NFP), which indicates the level of optimization achieved by the fuzzer during the fuzzing session. This metric allows us to evaluate how effectively the fuzzer generates and uses different fuzzing payloads to maximize code coverage and vulnerability detection.

Table 4 Rule-based fuzzer performance for specific vulnerabilities

Vulnerability	TP	TN	FP	FN	Acc.	Prec.	Rec.	Tot.
SQLI	125	47	56	0	0.75	0.69	1	228
PT	117	69	34	0	0.85	0.77	1	220
XSS	54	48	55	1	0.65	0.49	0,.98	158

Table 5 Enhanced rule-based fuzzer performance metrics for XSS vulnerabilities

Vulnerability	TP	TN	FP	FN	Acc.	Prec.	Rec.	Tot.
XSS	54	103	0	1	0,99	1	0,98	158

6.2 Rule-based fuzzer performance

The rule-based fuzzer was tested on the benchmark, and performance metrics were collected. Table 4 shows the metrics for each vulnerability type. It is possible to observe that the fuzzer has high accuracy on SQLi and PT vulnerabilities. The number of false positives is higher for XSS. The explanation for this behavior is that the rule-based fuzzer checks if an XSS string payload is reflected without analyzing the so-called "reflection context" [40].

We address this problem by adding a vulnerability rule that leverages a headless browser to check for the actual execution of JavaScript code, hence improving the discovery of cross-site scripting vulnerabilities, in the same way as indicated in [41]. Results with the additional rule are shown in Table 5.

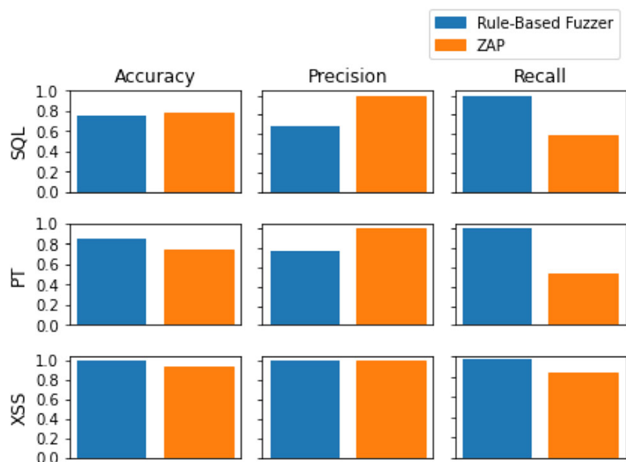

Fig. 8 Comparison between Rule-Based Fuzzer and ZAP

6.3 Zed attack proxy comparison

OWASP Zed Attack Proxy (ZAP) [42] is a popular open-source web application security scanner that is widely used by professional penetration testers to identify vulnerabilities in web applications. It is considered to be a reliable alternative to commercial solutions, as it offers comparable performance. This has been confirmed in Chen's dynamic application security testing solutions comparison [43], which recognized ZAP as an effective tool in this domain.

Figure 8 shows a comparison between the Rule-Based Fuzzer and Zed Attack Proxy performance metrics for each vulnerability type.

The results show a similar accuracy for SQL injection vulnerabilities and better accuracy of the rule-based fuzzer for both XSS and PT vulnerabilities, although with lower precision. Concerning recall, the rule-based fuzzer performs better than ZAP in the case of SQL injection. Figures are comparable for the XSS case. We perform worse than ZAP when it comes to PT vulnerabilities. With reference to the above results, we observe that precision and recall could both be improved by tuning the vulnerability rules. The high level of accuracy, on the other hand, proves the effectiveness of the chosen payloads, as it would not be possible to trigger the vulnerabilities without proper payloads. With the current settings, the rule-based fuzzer outperforms ZAP in the case of cross-site scripting vulnerabilities. Tables 6 and 7 show the results in more detail.

6.4 Strengths and limitations of the rule-based fuzzer

The performance of the rule-based fuzzer depends on the number and type of payloads used during fuzzing. A high number of payloads increases the vulnerability discovery rate but reduces efficiency. Table 8 shows that performance depends on the number of used payloads (NFP parameter in the table).

The table shows that performance metrics improve as the number of payloads increases. A vulnerability is discovered when a payload alters the normal process of the target application by revealing a security flaw. Hence, the fuzzer has a better chance of triggering abnormal application behaviors if more payloads are used. However, more payloads lead to a decrease in efficiency.

Figure 9 and Table 9 show the relative trend between the number of payloads, the accuracy of the scanner, and the number of requests. For example, accuracy slightly increases if the number of payloads is equal to 143, but the number of executed requests doubles compared to when we leverage just

Table 6 Comparison between rule-based fuzzer and ZAP: detailed figures

	Rule-based fuzzer				ZAP				Total
	TP	TN	FN	FP	TP	TN	FN	FP	
SQLi	125	47	0	56	74	103	51	0	228
PT	117	69	0	34	63	103	54	0	220
XSS	54	103	1	0	46	103	9	0	158

Table 7 Accuracy, precision and recall of rule-based fuzzer and ZAP

	Rule-based fuzzer			ZAP		
	SQLi	PT	XSS	SQLi	PT	XSS
Accuracy	0.75	0.85	0.99	0.77	0.75	0.94
Precision	0.69	0.77	1	1	1	1
Recall	1	1	0.98	0.59	0.53	0.83

Table 8 Rule-Based Fuzzer performance improvements for a greater number of payloads

NFP	TP	TN	FP	FN	Accuracy	Precision	Recall
32	212	49	54	85	0.62	0.79	0.71
64	245	49	54	52	0.73	0.81	0.82
77	269	49	54	28	0.79	0.83	0.90
143	296	44	59	1	0.85	0.83	0.99

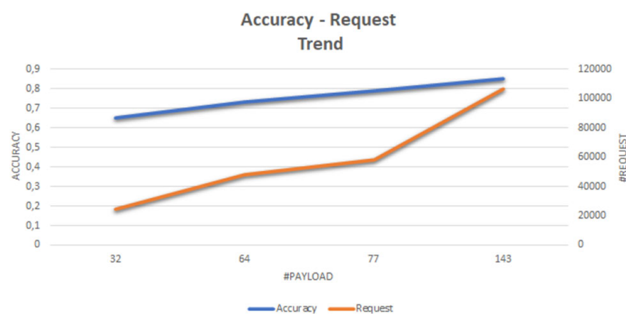

Fig. 9 Accuracy-Request Trend

Table 9 Accuracy and number of requests versus the number of Payloads

Payload	Accuracy	No. requests
32	0.65	24,128
64	0.73	48,256
77	0.79	58,058
143	0.85	106,642

77 payloads. The increase in accuracy is 6% in such a case. The extensibility of the rule-based fuzzer allows changing the number of payloads by simply adding attack strings in the configuration file of the Analyzer module. The identification of the most suitable trade-off between the number of payloads and the scanner's accuracy does indeed represent an integral part of the rule-based fuzzer's instrumentation process.

Generally, it is important to underline a few intrinsic limitations of a rule-based approach. The vulnerability knowledge base has been implemented through the process described in Sect. 5. It is clearly possible to extend the process to other training scenarios and increase the system's performance by adding new rules and assertions. The approach depends on the effectiveness of the knowledge base and requires a continuous update. As new attacks are devised on a daily basis, keeping the pace of required updates to the knowledge base can become very challenging. Another problem associated with the quality of the information contained in the knowledge base concerns the selection of "good-enough" fuzzing strings for the triggering of the entire set of target vulnerabilities. With reference to the points raised above, in the conclusions section, we will propose some ideas to overcome the "static nature" of the rule-based approach.

7 Optimization strategies

The previous section analyzes the performance of the rule-based fuzzer and highlights the limitations of this approach. In this section, we will first discuss a couple of general performance enhancement criteria. Then, our focus will shift to the specific topic of payload optimization. Finally, we will introduce a few additional performance metrics.

7.1 Performance enhancements

As observed in Sect. 6.4, the performance metrics improve when a larger number of payloads are used. This behavior can be attributed to the use of more "feature classes" when a greater number of payloads are employed. We explain the concept of "feature class" in Sect. 7.2. However, increasing the number of payloads does not necessarily increase the precision metric, as it is influenced by the number of false positives. To mitigate the occurrence of false positives and reduce incorrect results, it is possible to implement simple criteria that effectively decrease their number.

Performance Enhancement Criterion 1:
Compare the observations obtained by sending a valid payload with those obtained with the fuzzed payload.

This criterion aims to reduce the number of false positives by analyzing the response content, particularly for rules involving the analysis of strings. For example, consider a Path Traversal (PT) rule that aims to detect the presence of the "root" string in the response content, which would indicate that the web server may have executed the "/etc/passwd" operating system command. However, it is important to note that the "root" string may also appear in the response content, especially within code comments, as illustrated in Listing 5.

If the knowledge base rules check for the presence of certain words in the observed valid response, it can help reduce the number of false positives. However, it would be more effective if the observations did not contain the words used to trigger a vulnerability in the first place. Continuing with the previous example, the "passwd" file often contains the word "sbin" in its content. This word is very specific and less likely to be found in source code comments or other non-relevant areas. This forms the basis for the second criterion.

Performance Enhancement Criterion 2:
When selecting a string presence in the content of an HTTP response to trigger a rule and there's a choice between two different words, opt for the more discriminant word, i.e., the one that minimizes the chances of being found in the content of a valid request.

In Sect. 5.5, we emphasize the importance of considering the payload type to trigger the appropriate rules. This approach can also be extended to analyze the type of payloads being used and prevent the activation of irrelevant rules. For instance, when employing SQL payloads to detect SQL injection vulnerabilities, it is beneficial to distinguish between error-based and time-based payloads.

The rules within the oracle can evaluate conditions depending on the specific subcategory. For instance, the "anomalous time" check would not be triggered unless a "time-based" SQL payload is used.

Performance Enhancement Criterion 3:
Whenever possible, employ more fine-grained payload types and minimize the number of activated rules.

By implementing these straightforward rules, it becomes possible to enhance the performance of the rule-based fuzzer. In the benchmark detailed in Sect. 6, we can observe the improved statistics in Table 10, clearly illustrating the desired performance enhancement.

```
<p>
<i>
  <code>install-noconsolehelper.sh</code>
</i>.
Both scripts perform sanity checks to ensure OProfile is installed and that
<code>opxml</code>, a C++ program required to interface with OProfile, exists and can be run.
The difference is in how root authentication with the plug-in is set up.
<i><code>install.sh</code></i> uses the pluggable authentication modules (PAM) mechanism.
This is the default and recommended method for root authentication. When an OProfile task is
required, you will be presented with this dialog to enter the root password:
...
<a href="Troubleshooting.html#Troubleshooting">Troubleshooting</a> section.

</p>
```

Listing 5: Source code with no path traversal vulnerability and root string in the content. Ref.: https://searchcode.com/codesearch/raw/687277589/

Table 10 Increased rule-based fuzzer performance with optimization criteria

Vulnerability	Acc.	Prec.	Rec.
SQLI	0.98	0.96	1
PT	0.99	0.97	1
XSS	0.99	1	0.98

7.2 Payloads optimizations

As outlined in Sect. 6.4, augmenting the number of payloads correlates with improved performance metrics. This relationship stems from the payloads' capacity to elicit the observations necessary for identifying vulnerabilities. Nevertheless, it is feasible to establish optimization techniques aimed at reducing the number of payloads while maintaining the same level of vulnerability detection performance. In this section, we expound on these strategies, which can be categorized into two groups:

1. *Payload Feature Optimization*: these optimizations focus on enhancing the effectiveness of individual payloads by refining their features.
2. *Optimization through Web Application Enumeration*: these optimizations revolve around the process of enumerating a web application to pinpoint vulnerabilities more efficiently.

7.2.1 Payload feature optimization

In a broader context, this category of optimization strategies can be subdivided into the following more specific subcategories:

- *Semantic Payload Generation*: instead of generating a large number of random payloads, focus on generating payloads with specific semantic features that are more likely to trigger vulnerabilities. For instance, for SQL injection testing, you can create payloads that contain common SQL keywords or syntax patterns known to be effective in revealing vulnerabilities;
- *Dynamic Payload Generation*: develop a dynamic payload generation mechanism that adapts payloads based on the application's responses. Start with simple payloads and gradually increase complexity as you gather more information about the application's behavior. This can help reduce the number of payloads needed while maintaining effective testing;
- *Payload Mutation Strategies*: implement strategies for mutating existing payloads intelligently. Rather than creating entirely new payloads, modify existing ones by changing specific parts of the payload. This can lead to a more efficient use of payloads.

The best payload set aims to minimize the number of payloads having different features. Through the proven experience of web application penetration testers, we try to define several features for each vulnerability. However, this approach is generic and can be further extended to any type of feature classification and improved by adding further features.

The general observation is that each payload "stimulates" the web application in various ways, and each payload possesses specific characteristics that can be shared with other payloads. For instance, an SQL injection can be executed using a 'sleep' instruction, a single quote character, or a double quote character. Moreover, the payload might contain the valid payload string or replace it. Each condition can be considered a feature or characteristic of a payload, and these features can have categorical values. As a result, it is reason-

able to expect that two payloads with identical characteristics will trigger the same anomalous observations. We can, hence, define features for payloads that allow us to categorize them into similar classes. A 'feature class' consists of payloads that share the same set of features.

Payload Optimization Criterion 1:

For each feature class, extract a single payload. The optimal payload set should include one payload from each feature class. The goal of the best payload set is to minimize the number of payloads with different features.

We draw from the extensive experience of web application penetration testers to define numerous features for each vulnerability.
However, this approach is versatile and can be expanded to encompass any feature classification while also benefiting from the addition of further features.

XSS features For cross-site scripting vulnerabilities, it is possible to identify several features. To trigger the vulnerability, the injected payload must be reflected within an executable JavaScript code without causing errors. The valid payload is contingent upon the reflection context, which is the location where the input is reflected. The payload that 'triggers' the vulnerability is also influenced by the input sanitization methods employed by the web application to mitigate web attacks. In fact, for each defensive mechanism, it is possible to modify the payload to circumvent it [44]. Table 11 presents the list of analyzed features.

SQLi features

For SQL injection vulnerabilities, it is essential to categorize the input type. For instance, error-based or blind-based SQL injection attacks aim to generate an SQL syntax error, which would either display an SQL error message or return an error status code. In contrast, a time-based SQL injection attack attempts to execute an SQL 'sleep' command, creating a time delay in the web application that can be analyzed. The oracle can be fine-tuned by distinguishing between payload types (SQL or time-based SQL). Specifically, if the payload used is not time-based, a response delay will not be evaluated as an SQL injection condition.

Another critical feature to consider is the type of quotation mark used in the backend SQL queries. When a web application is vulnerable to SQL injection, the injected payload is inserted into an SQL query that retrieves data from the database. Typically, the input is injected into a 'WHERE' clause and can be enclosed in quotation marks, such as single or double quotes. As previously mentioned, an error-based SQL injection vulnerability can be exposed by causing an SQL syntax error in the web application. Therefore, if the injected payload is enclosed in double quotes, a payload that triggers an SQL syntax error must contain a double quote[4].

As this information is not usually deducible through information gathering,[5] a comprehensive payload set should encompass all possible quotation marks. Additionally, an important feature to consider is the underlying Database Management System (DBMS) in use. Each DBMS employs a specific SQL dialect with different instructions. For instance, PostgreSQL uses the `pg_sleep()` instruction to induce sleep, while MySQL uses `SELECT SLEEP()`. For each dialect, it is possible to define multiple payloads. If N represents the number of dialects under examination, the number of payloads required to cover all possible databases is $M \times N$. Payload reduction can be achieved by employing enumeration techniques as described in Sect. 7.2.2.

As with XSS vulnerabilities, the analysis of the sanitization techniques employed by the web application is crucial for SQL injection vulnerabilities [45]. Table 12 provides a summary of valuable features for SQL injection vulnerabilities.

PT features

To trigger a path traversal vulnerability, the injected payload must point to a readable and existing file. This type of vulnerability occurs due to flaws in a web application that allow arbitrary file reading. The specific file that constitutes a valid payload depends on the underlying operating system and web server interpreter. For instance, a valid file for PHP might be `login.php`, whereas for a Windows system, it could be `c:\Windows\System32\Drivers\etc\hosts`. In the context of path traversal vulnerabilities, the presence or absence of sanitization techniques that block special characters such as dots, backslashes, and slashes are noteworthy features. These features can significantly influence the variety of payloads needed to comprehensively test for this vulnerability [46]. Table 13 provides an overview of the analyzed features.

7.2.2 Optimization through web application enumeration

Payload reduction through Web Application Enumeration can happen in different ways:

[4] For example, if the `<username>` payload is used to create the following SQL query: `SELECT * from USER WHERE name = "<username>"`, sending `test"` as the payload would generate an SQL query with three double quotes, causing a syntax error that can be analyzed to discover the vulnerability.

[5] The quotation mark can be ascertained by inspecting the backend source code. This information may be obtained through a vulnerability that exposes the source code or by employing a grey-box fuzzing approach. In such cases, the payload set can be streamlined by exclusively utilizing the correct quotation mark.

Table 11 XSS features

Feature	Descr.	Possible values
Reflection context	Where the string is reflected	In body, in JS code, in attribute, in comment, in event
Filtering	Which strings are filtered	Quote, double quote, script, img, tag bracket, backtick
Escaping	Which strings are escaped	Quote, double quote, script, img, tag bracket, backtick
Encoding	Which strings are encoded	Quote, double quote, script, img, tag bracket, backtick

Table 12 SQLi features

Feature	Descr.	Possible values
DMBS	The used DBMS	MySQL, PostgreSQL, MSSQL
Payload type	The used payload	Time-based SQL, SQL
Filtering	Which strings are filtered	Quote, double quote, SELECT, space
Escaping	Which strings are escaped	Quote, double quote, SELECT, space
Encoding	Which strings are encoded	Quote, double quote, SELECT, space

Table 13 PT features

Feature	Descr.	Possible values
OS	The underlying operating system	Linux, Windows
The file type	Which file type is used	OS file, PHP file, Application Server file, etc
Filtering	Which strings are filtered	Dot, backslash, slash
Escaping	Which strings are escaped	Dot, backslash, slash
Encoding	Which strings are encoded	Dot, backslash, slash

- *Application Profiling*: before conducting the fuzzing process, profile the target application thoroughly to identify its attack surface, input vectors, and potential vulnerabilities. This information can guide the selection of payloads, reducing the need for a large number of generic payloads;
- *Input Enumeration*: enumerate and catalog all possible input points in the web application, including input fields, URLs, and headers. Prioritize testing these inputs over less critical ones;
- *Response Analysis*: analyze the application's responses to initial requests to gain insights into its behavior. Identify patterns or anomalies that can help you craft more targeted payloads.

By incorporating these strategies into the fuzzing process, one can significantly reduce the number of payloads required while maintaining a high level of effectiveness in detecting vulnerabilities. This optimization can lead to more efficient and focused testing, especially in situations where generating a massive number of payloads is resource-intensive or time-consuming.

The effectiveness of payloads is also contingent upon the specific web application environment. For example, a path traversal payload designed for a Windows-based system would be ineffective against a Linux-based one. This underscores the importance of tailoring payloads to suit the target environment. It is worth noting that web application scanners are generally not optimized for environment analysis and payload selection. Instead, they typically employ a predefined set of payloads intended to trigger known vulnerabilities. Recent works [47, 48] have demonstrated the feasibility of formalizing penetration testing activities in terms of hacking goals. We have introduced an algorithm and a generic framework for integrating various actions and attacks, enabling the discovery of web vulnerabilities through an offensive approach. An effective method for improving web application security is to include enumeration actions before the fuzzing phase, utilizing the methodology we have proposed.

Through this approach, it is possible to infer the web application environment and consequently reduce the number of payloads. Once the enumeration phase is completed, attacks can be launched against the web application.

XSS footprinting

For cross-site scripting vulnerabilities, it is crucial to verify two conditions:

1. the sent payload is reflected.
2. the "reflection context" is an executable one.

Table 14 Number of payloads for XSS detection

Scanner	NP
ZAP	52
Non optimized rule-based fuzzer	15
Rule-based fuzzer with minimized payloads	13

Table 15 Number of payloads for SQL injection detection

Scanner	NP
ZAP	157
Non optimized rule-based fuzzer	67
Rule-based fuzzer with minimized payloads	29

In [40], we introduced the concept of the reflection context and demonstrated its utility in optimizing the number of requests. A similar approach can be applied in this work to identify cross-site scripting vulnerabilities. By analyzing where the input is reflected when a request is sent, it becomes possible to reduce the number of payloads to just a single one. However, several requests may need to be sent to properly understand the reflection context.

SQL and PT footprinting

If the used DBMS is discovered, it is possible to divide the number of payloads by excluding those that target other SQL dialects. Information Conflict of interest vulnerabilities can reveal the used DBMS and facilitate this optimization. For example, the presence of a `PHPinfo` file can disclose essential information like the underlying DBMS and the operating system in use. Therefore, by detecting the used DBMS, it is possible to exclude unnecessary payloads.

In the case of path traversal vulnerabilities, it is crucial to identify underlying systems, such as the operating system or the webserver running the web application. This knowledge allows for the exclusion of payloads designed to exploit vulnerabilities in different systems. For instance, if the underlying operating system is Linux, a payload attempting to exploit a path traversal vulnerability by reading the content of `C:\Windows\System32\drivers\etc\hosts` is ineffective. Therefore, discovering the underlying system permits payload optimization without compromising performance metrics.

7.2.3 Payload optimization: summary results

Tables 14, 15, and 16 display the quantity of payloads employed by ZAP, as well as by our rule-based fuzzer, both with and without the payload number reduction strategies detailed in Sect. 7.2.2.

As can be observed, the optimization strategies resulted in a 13% reduction in the number of payloads used for

Table 16 Number of payloads for PT detection

Scanner	NP
ZAP	18
Non optimized rule-based fuzzer	59
Rule-based fuzzer with minimized payloads	5

Table 17 Comparison table: language support and vulnerability coverage

Work	Language	XSS	SQLi	PT
Our work	Any	✓	✓	✓
[50]	JAVA	✓	✓	✓
[51]	Any	✓	✓	✓
[52]	PHP	✓		
[53]	PHP	✓	✓	✓
[54]	Any		✓	
[55]	Any		✓	
[56]	PHP	✓		
[57]	Any	✓	✓	✓
[58]	PHP	✓	✓	✓
[59]	PHP	✓	✓	✓
[60]	PHP	✓		
[61]	PHP	✓	✓	✓
[62]	PHP	✓		
[63]	Java	✓		
[64]	Java		✓	
[65]	Any	✓	✓	
[66]	Java		✓	
[67]	Any		✓	
[68]	Any		✓	
[69]	Any		✓	
[70]	Java	✓		
[71]	Java		✓	
[72]	Any	✓	✓	
[73]	PHP	✓		
[74]	Any	✓	✓	
[75]	Java		✓	
[76]	Java	✓		✓
[77]	Java		✓	
[78]	Java	✓	✓	
[79]	Any	✓	✓	
[80]	PHP	✓	✓	
[81]	PHP	✓		
[18]	PHP	✓	✓	
[82]	Any	✓	✓	
[83]	PHP	✓		
[84]	PHP	✓	✓	
[85]	Any		✓	
[86]	Any		✓	

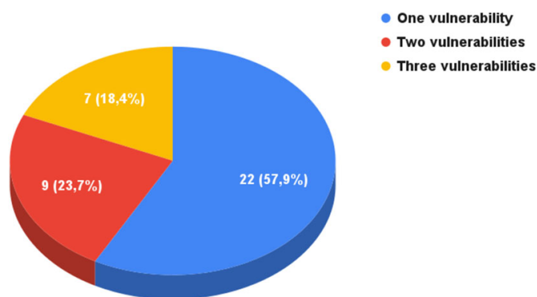

Fig. 10 Scanners and vulnerability coverage

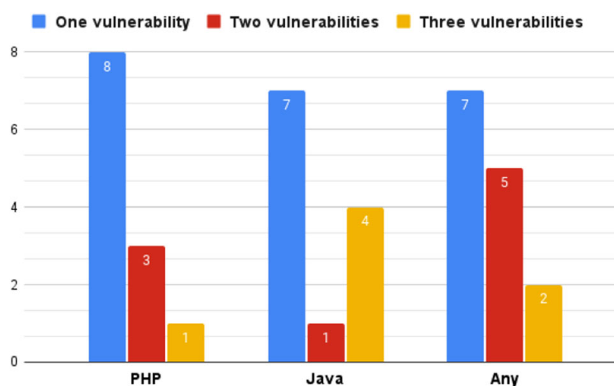

Fig. 11 Vulnerability coverage per language

XSS detection, a 56% reduction in the number of payloads used for SQL injection detection, and a remarkable 91% reduction for path traversal detection. This indicates that the non-optimized rule-based fuzzer's input payload set contained numerous inputs with similar features, as discussed in Sect. 7.2.

7.3 Additional performance metrics

In order to evaluate the effectiveness of the optimization strategies and enhance the assessment of scanners, it is possible to include additional performance metrics. As noted earlier, improving a scanner's performance involves reducing the number of payloads while maintaining or even enhancing its effectiveness. Two additional vital metrics are the scanner's vulnerability coverage and its compatibility with various source code languages. Notably, many scanners in the literature are specialized for a single vulnerability type or can only be applied to web applications developed in specific programming languages [49].

In summary, two new performance metrics can be defined:

- **Vulnerability Coverage (VC)**: VC measures the number of vulnerabilities that a scanner is capable of identifying;
- **Language Support (LS)**: LS assesses the range of programming languages supported by the scanner.

We conducted an analysis of vulnerability coverage and language support based on 38 studies included in Zhang et al.'s 2021 survey [49]. The results are presented in Table 17.

Figure 10 illustrates that 57,9% of the scanners mentioned in the survey's references primarily focus on a single vulnerability, while only 18,4% of them cover all three different types of vulnerabilities. Among these, only two also happen to be language-independent (see Fig. 11).

Unfortunately, many works do not make their source code publicly available, making it challenging to replicate experiments or build upon the authors' approaches. In contrast, we have made our source code publicly accessible to facilitate improvement and extension of our approach Table 18 presents a summary of the key performance metrics, including Accuracy, Precision, Recall, and Number of Payloads, attained by our fuzzer before and after implementing the discussed optimizations.[6]

Table 18 Comparison after the application of optimization steps

	Rule-based fuzzer				Optimized rule-based			
	Acc.	Prec.	Recall	NP	Acc.	Prec.	Recall	NP
SQLi	0.74	0.69	1	67	0.98	0.96	1	29
PT	0.85	0.77	1	59	0.99	0.07	1	5
XSS	0.99	1	0.98	15	0.99	1	0.98	1[7]

By analyzing the reflection context, it is possible to define the payload dynamically. Obviously, the footprinting step requires several requests

8 Conclusions

In this work, we have introduced a rule-based fuzzer designed to detect web application injection vulnerabilities, with a particular focus on cross-site scripting, SQL Injection, and Path Traversal. It is worth noting that our proposed approach can be adapted to identify various types of input-based vulnerabilities. We conducted a comprehensive performance analysis comparing our system with Zed Attack Proxy (ZAP), one of the most widely used web application security scanners. Our results demonstrate that the rule-based approach can yield comparable outcomes to those achieved with ZAP. Additionally, we discussed how optimizing payloads, observations, and rules can enhance the performance of our rule-based fuzzer.

We have identified the number of used payloads as a crucial performance metric and have shown that it can be reduced through specific optimization criteria. Nevertheless, there is room for further payload reduction through the exploration of alternative approaches.

[6] https://github.com/NS-unina/Rule-Based-Fuzzer/tree/master.

As part of our future work, we intend to concentrate on devising a payload optimization strategy to maximize vulnerability coverage while minimizing the number of payloads used. Drawing from past experience [40], we believe that an approach rooted in Reinforcement Learning (RL) can be instrumental in achieving this objective. With RL, we can train an agent to detect all vulnerabilities within a predefined environment while minimizing payload usage.

In recent years, Large Language Models (LLMs) have made substantial contributions to natural language processing (NLP) [87]. Several contemporary approaches are investigating the security implications of generating source code using such models [88, 89]. They aim to determine whether these models can be leveraged to identify and rectify security vulnerabilities in source code [90–92]. While these approaches typically operate in a white-box context, there is potential for our work to benefit from the adoption of LLMs. As elaborated in Sect. 7, optimization criteria encompass tasks such as application profiling, input enumeration, and response analysis. These are areas where Natural Language Processing techniques could be applied effectively. In future research, we plan to explore this avenue to enhance our work.

The results we have presented in this work are contingent on the test suite we used. Although some research endeavors have attempted to gauge the effectiveness of test suites for evaluating web scanners, no formalized test suite that guarantees complete vulnerability coverage of a web scanner [49] currently exists. In future endeavors, we intend to delve into this issue, aiming to formalize performance metrics for web application vulnerability test suites. This will involve delineating the limitations of existing platforms and proposing enhanced alternatives.

Our forthcoming efforts will also be directed toward enhancing the Oracle module by introducing assertions and vulnerability observations. In this context, an evolved rule-based approach may harness a reinforcement learning model to automatically generate rules. The modular architecture of our platform facilitates the collection of requests from a training environment, replication within a specific test environment, and performance evaluation.

We are optimistic that our work will serve as a valuable resource for security researchers seeking to explore novel fuzzing approaches for identifying input-handling vulnerabilities.

Funding Open access funding provided by Universitá degli Studi di Napoli Federico II within the CRUI-CARE Agreement.

Declaration

Conflict of interest Declaration The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Singh, N., Meherhomji, V., Chandavarkar, B.: Automated versus manual approach of web application penetration testing. In: 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT). IEEE, pp. 1–6 (2020)
2. Aydos, M., Aldan, Ç., Coşkun, E., Soydan, A.: Security testing of web applications: a systematic mapping of the literature. J. King Saud Univ. Comput. Inf. Sci. **34**, 6775–6792 (2021)
3. The owasp testing guide (2022). [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/. Accessed 01 Feb 2021
4. Path traversal. https://owasp.org/www-community/attacks/Path_Traversal. Accessed 12 Feb 2022
5. Kowalski, R.: Predicate logic as programming language. In: IFIP Congress, vol. 74, pp. 569–544 (1974)
6. Horn, A.: On sentences which are true of direct unions of algebras. J. Symbol. Log. **16**(1), 14–21 (1951)
7. Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artif. Intell. **2**(3–4), 227–260 (1971)
8. Apt, K.R.: Logic programming. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), vol. 1990, pp. 493–574 (1990)
9. Kok, J.N.: Specialization in logic programming: from horn clause logic to prolog and concurrent prolog. In: Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems). Springer, pp. 401–413 (1989)
10. A concise introduction to prolog. https://www.cis.upenn.edu/matuszek/Concise. Accessed 03 Mar 2021
11. Carlucci Aiello, L., Massacci, F.: Verifying security protocols as planning in logic programming. ACM Trans. Comput. Log. **2**(4), 542–580 (2001)
12. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Security protocols verification in abductive logic programming: a case study. In: International Workshop on Engineering Societies in the Agents World. Springer, pp. 106–124 (2005)
13. Barker, S.: Data protection by logic programming. In: International Conference on Computational Logic. Springer, pp. 1300–1314 (2000)
14. Zech, P., Felderer, M., Breu, R.: Security risk analysis by logic programming. In: International Workshop on Risk Assessment and Risk-Driven Testing. Springer, pp. 38–48 (2013)
15. Vemuri, R., Kalyanaraman, R.: Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. IEEE Trans. Very Large Scale Integr. Syst. **3**(2), 201–214 (1995)
16. Denney, R.: Test-case generation from prolog-based specifications. IEEE Softw. **8**(2), 49–57 (1991)

17. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. Theory Pract. Logic Program. **10**(4–6), 659–674 (2010)
18. Zech, P., Felderer, M., Breu, R.: Knowledge-based security testing of web applications by logic programming. Int. J. Softw. Tools Technol. Transf. **21**(2), 221–246 (2019)
19. Boehme, M., Cadar, C., Roychoudhury, A.: Fuzzing: challenges and reflections. IEEE Softw. **38**(3), 79–86 (2021)
20. McNally, R., Yiu, K., Grove, D., Gerhardy, D.: Fuzzing: the state of the art. Defence Science and Technology Organisation Edinburgh (Australia), Technical report (2012)
21. Pham, V.-T., Bohme, M., Santosa, A., Caciulescu, A., Roychoudhury, A.: Smart greybox fuzzing. IEEE Trans. Softw. Eng. **47**(9), 1980–1997 (2021)
22. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: a survey. IEEE Trans. Softw. Eng. **47**, 2312–2331 (2019)
23. Sinha, S.: The exploration-exploitation dilemma: a review in the context of managing growth of new ventures. Vikalpa **40**(3), 313–323 (2015)
24. Woo, M., Cha, S. K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 511–522 (2013)
25. Berry, D.A., Fristedt, B.: Bandit Problems: Sequential Allocation of Experiments. In: Monographs on Statistics and Applied Probability. Chapman and Hall, London, vol. 5, no. 71-87, pp. 7–7 (1985)
26. Householder, A.D., Foote, J.M.: Probability-based parameter selection for black-box fuzz testing. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Technical report (2012)
27. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
28. Duchene, F., Rawat, S., Richier, J.-L., Groz, R.: Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, pp. 37–48 (2014)
29. Appelt, D., Nguyen, C. D., Briand, L. C., Alshahwan, N.: Automated testing for SQL injection vulnerabilities: an input mutation approach. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 259–269 (2014)
30. Khalid, M.N., Farooq, H., Iqbal, M., Alam, M. T., Rasheed, K.: Predicting web vulnerabilities in web applications based on machine learning. In: International Conference on Intelligent Technologies and Applications. Springer, pp. 473–484 (2018)
31. Liu, Z., Fang, Y., Huang, C., Xu, Y.: Mfxss: an effective xss vulnerability detection method in JavaScript based on multi-feature model. Comput. Secur. **124**, 103015 (2023)
32. Song, X., Zhang, R., Dong, Q., Cui, B.: Grey-box fuzzing based on reinforcement learning for xss vulnerabilities. Appl. Sci. **13**(4), 2482 (2023)
33. Altulaihan, E.A., Alismail, A., Frikha, M.: A survey on web application penetration testing. Electronics **12**(5), 1229 (2023)
34. Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: software metrics vs text mining. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 23–33 (2014)
35. Li, L., Dong, Q., Liu, D., Zhu, L.: The application of fuzzing in web software security vulnerabilities test. In: 2013 International Conference on Information Technology and Applications. IEEE, pp. 130–133 (2013)
36. Mitmproxy is a free and open source interactive https proxy. https://mitmproxy.org/. Accessed 12 Feb 2022
37. What is a transparent proxy: client vs. server side use cases: Imperva (2020). https://www.imperva.com/learn/ddos/transparent-proxy/. Accessed 10 Jan 2022
38. Armstrong, J.: Computer Logic, Testing and Verification. By Paul Roth, vol. 90. Taylor & Francis, London (1983)
39. Sectooladdict, Sectooladdict/wavsep: the web application vulnerability scanner evaluation project. https://github.com/sectooladdict/wavsep. Accessed 01 June 2021
40. Caturano, F., Perrone, G., Romano, S.: Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment". Comput. Secur. **103**, 102204 (2021)
41. Lv, C., Zhang, L., Zeng, F., Zhang, J.: Adaptive random testing for xss vulnerability. In: Proceedings—Asia-Pacific Software Engineering Conference, APSEC, vol. 2019–December, pp. 63–69 (2019)
42. Bennetts, S.: Owasp zed attack proxy (2013). https://owasp.org/www-project-zap/. Accessed 01 Mar 2021
43. Chen, S.: Wavsep 2017/2018—evaluating dast against pt/sdl challenges (1970). http://sectooladdict.blogspot.com/2017/11/wavsep-2017-evaluating-dast-against.html. Accessed 03 Mar 2021
44. OWASP, Xss filter evasion cheat sheet (2008). https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html. Online. Accessed 4 Sep 2023
45. Swigger, P.: SQL injection bypassing common filters (2023). https://portswigger.net/support/sql-injection-bypassing-common-filters. Online. Accessed 4 Sep 2023
46. Google, Advanced obfuscation path traversal (2023). https://code.google.com/archive/p/teenage-mutant-ninja-turtles/wikis/AdvancedObfuscationPathtraversal.wiki. Online. Accessed 4 Sep2023
47. Caturano, F., Perrone, G., Romano, S. P.: Hacking goals: a goal-centric attack classification framework. In: Testing Software and Systems: 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9–11, 2020, Proceedings. Springer, Berlin, pp. 296-301 (2020). [Online]. https://doi.org/10.1007/978-3-030-64881-7_19
48. Auricchio, N., Cappuccio, A., Caturano, F., Perrone, G., Romano, S.P.: An automated approach to web offensive security. Comput. Commun. **195**, 248–261 (2022)
49. Zhang, B., Li, J., Ren, J., Huang, G.: Efficiency and effectiveness of web application vulnerability detection approaches: a review. ACM Comput. Surv. (2021). https://doi.org/10.1145/3474553
50. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: Security slicing for auditing common injection vulnerabilities. J. Syst. Softw. **137**, 766–783 (2018). https://doi.org/10.1016/j.jss.2017.02.040
51. Medeiros, I., Neves, N., Correia, M.: Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Trans. Reliab. **65**(1), 54–69 (2016). https://doi.org/10.1109/tr.2015.2457411
52. Yan, X., Ma, H., Wang, Q.: A static backward taint data analysis method for detecting web application vulnerabilities. In: 2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN). IEEE (may 2017) [Online]. https://doi.org/10.1109/iccsn.2017.8230288
53. Noman, M., Iqbal, M., Talha, M., Jain, V., Mirza, H., Rasheed, K.: Web unique method (WUM): an open source blackbox scanner for detecting web vulnerabilities. IJACSA (2017). https://doi.org/10.14569/ijacsa.2017.081254
54. Awang, N.F., Manaf, A.A.: Automated security testing framework for detecting SQL injection vulnerability in web application. In: Jahankhani, H., Carlile, A., Akhgar, B., Taal, A., Hessami, A.G., Hosseinian-Far, A. (eds.) Global Security, Safety and Sustainability: Tomorrow's Challenges of Cyber Security, pp. 160–171. Springer, Cham (2015)
55. Ciampa, A., Visaggio, C. A., Penta, M. D.: A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In: Proceedings of the 2010 ICSE Workshop on Software

Engineering for Secure Systems. ACM (may 2010) [Online]. https://doi.org/10.1145/1809100.1809107
56. Gupta, M.K., Govil, M. C., Singh, G., Sharma, P.: Xssdm: towards detection and mitigation of cross-site scripting vulnerabilities in web applications. In: 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2010–2015 (2015)
57. Akrout, R., Alata, E., Kaaniche, M., Nicomette, V.: An automated black box approach for web vulnerability identification and attack scenario generation. J. Braz. Comput. Soc. **20**(1), 4 (2014). https://doi.org/10.1186/1678-4804-20-4
58. Medeiros, I., Neves, N., Correia, M.: Dekant: a static analysis tool that learns to detect web application vulnerabilities. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery (2016), pp. 1–11. [Online]. https://doi.org/10.1145/2931037.2931041
59. Jensen, T., Pedersen, H., Olesen, M.C., Hansen, R.R.: Thaps: automated vulnerability scanning of php applications. In: Jøsang, A., Carlsson, B. (eds.) Secure IT Systems, pp. 31–46. Springer, Berlin (2012)
60. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 13th International Conference on Software Engineering—ICSE '08. ACM Press (2008) [Online]. https://doi.org/10.1145/1368088.1368112
61. Shar, L.K., Briand, L.C., Tan, H.B.K.: Web application vulnerability prediction using hybrid program analysis and machine learning. Trans. Dependable Secure Comput. **12**(6), 688–707 (2015). https://doi.org/10.1109/tdsc.2014.2373377
62. Gupta, S., Gupta, B.B.: XSS-SAFE: a server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. Arab. J. Sci. Eng. **41**(3), 897–920 (2015). https://doi.org/10.1007/s13369-015-1891-7
63. Shar, L.K., Tan, H.B.K.: Auditing the defense against cross site scripting in web applications. In: Proceedings of the International Conference on Security and Cryptography—Volume 1: SECRYPT, (ICETE 2010), INSTICC. SciTePress, pp. 505–511 (2010)
64. Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: Candid: dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans. Inf. Syst. Secur. (2010). https://doi.org/10.1145/1698750.1698754
65. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: Proceedings of the 17th Conference on Security Symposium, ser. SS'08. USA: USENIX Association, pp. 31–43 (2008)
66. Jang, Y.-S., Choi, J.-Y.: Detecting SQL injection attacks using query result size. Comput. Secur. **44**, 104–118 (2014)
67. Lei, L., Jing, X., Minglei, L., Jufeng, Y.: A dynamic SQL injection vulnerability test case generation model based on the multiple phases detection approach. In: 2013 IEEE 37th Annual Computer Software and Applications Conference, pp. 256–261 (2013)
68. Liu, L., Xu, J., Guo, C., Kang, J., Xu, S., Zhang, B.: Exposing SQL injection vulnerability through penetration test based on finite state machine. In: 2016 2nd IEEE International Conference on Computer and Communications (ICCC), pp. 1171–1175 (2016)
69. Liu, L., Xu, J., Yang, H., Guo, C., Kang, J., Xu, S., Zhang, B., Si, G.: An effective penetration test approach based on feature matrix for exposing SQL injection vulnerability. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). IEEE, [Online] (2016). https://doi.org/10.1109/compsac.2016.55
70. Ruse, M.E., Basu, S.: Detecting cross-site scripting vulnerability using concolic testing. In: 2013 10th International Conference on Information Technology: New Generations. IEEE [Online] (2013). https://doi.org/10.1109/itng.2013.97
71. Lee, I., Jeong, S., Yeo, S., Moon, J.: A novel method for SQL injection attack detection based on removing SQL query attribute values. Math. Comput. Model. **55**(1), 58–68 (2012)
72. Vithanage, N.M., Jeyamohan, N.: Webguardia: an integrated penetration testing system to detect web application vulnerabilities. In: 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), pp. 221–227 (2016)
73. Avancini, A., Ceccato, M.: Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. Inf. Softw. Technol. **55**(12), 2209–2222 (2013). https://doi.org/10.1016/j.infsof.2013.08.001
74. Shar, L.K., Beng Kuan Tan, H., Briand, L.C.: Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In: 013 35th International Conference on Software Engineering (ICSE), pp. 642–651 (2013)
75. Djuric, Z.: A black-box testing tool for detecting SQL injection vulnerabilities. In: 2013 Second International Conference on Informatics and Applications (ICIA). IEEE, [Online] (2013). https://doi.org/10.1109/icoia.2013.6650259
76. Kumar Singh, A., Roy, S.: A network based vulnerability scanner for detecting SQLI attacks in web applications. In: 2012 1st International Conference on Recent Advances in Information Technology (RAIT), pp. 585–590 (2012)
77. Wu, H., Gao, G., Miao, C.: Test SQL injection vulnerabilities in web applications based on structure matching. In: Proceedings of 2011 International Conference on Computer Science and Network Technology, vol. 2, pp. 935–938 (2011)
78. Li, N., Xie, T., Jin, M., Liu, C.: Perturbation-based user-input-validation testing of web applications. J. Syst. Softw. **83**(11), 2263–2274 (2010). https://doi.org/10.1016/j.jss.2010.07.007
79. Chen, J.-M., Wu, C.-L.: An automated vulnerability scanner for injection attack based on injection point. In: 2010 International Computer Symposium (ICS2010). IEEE, [Online] (2010). https://doi.org/10.1109/compsym.2010.5685537
80. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE [Online] (2008). https://doi.org/10.1109/sp.2008.22
81. Ahmed, M.A., Ali, F.: Multiple-path testing for cross site scripting using genetic algorithms. J. Syst. Archit. **64**, 50–62 (2016). https://doi.org/10.1016/j.sysarc.2015.11.001
82. Thome, J., Shar, L.K., Bianculli, D., Briand, L.: An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE Trans. Softw. Eng. Trans. Softw. Eng. **46**(2), 163–195 (2020). https://doi.org/10.1109/tse.2018.2844343
83. Gupta, M.K., Govil, M.C., Singh, G.: Text-mining and pattern-matching based prediction models for detecting vulnerable files in web applications. J. Web Eng. **17**(1–2), 028–044 (2018)
84. Agosta, G., Barenghi, A., Parata, A., Pelosi, G.: Automated security analysis of dynamic web applications through symbolic code execution. In: 2012 Ninth International Conference on Information Technology—New Generations. IEEE [Online] (2012). https://doi.org/10.1109/itng.2012.167
85. Ceccato, M., Nguyen, C.D., Appelt, D., Briand, L. C.: SOFIA: an automated security oracle for black-box testing of SQL-injection vulnerabilities. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM [Online] (2016). https://doi.org/10.1145/2970276.2970343
86. Zhang, L., Zhang, D., Wang, C., Zhao, J., Zhang, Z.: Art4sqli: the art of SQL injection vulnerability discovery. IEEE Trans. Reliab. **68**(4), 1470–1489 (2019)
87. A survey of large language models (2023)

88. Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M.A., Cordeiro, L.C.: A new era in software security: towards self-healing software via large language models and formal verification (2023)
89. He, J., Vechev, M.: Large language models for code: security hardening and adversarial testing (2023)
90. Yang, G., Dineen, S., Lin, Z., Liu, X.: Few-sample named entity recognition for security vulnerability reports by fine-tuning pre-trained language models. In: Wang, G., Ciptadi, A., Ahmadzadeh, A. (eds.) Deployable Machine Learning for Security Defense, pp. 55–78. Springer, Cham (2021)
91. Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Examining zero-shot vulnerability repair with large language models. In: IEEE Symposium on Security and Privacy (SP), vol. 2023, pp. 2339–2356 (2023)
92. Noever, D.: Can large language models find and fix vulnerable software? (2023)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.