university of groningen

faculty of science and engineering

An Exploration of Current Techniques in OWASP Vulnerability Detection and Improvement Opportunities

Andrei-Claudiu Veres

university of groningen / faculty of science and engineering

University of Groningen & TNO

An Exploration of Current Techniques in OWASP Vulnerability Detection and Improvement Opportunities

Internship Project

To fulfill the requirements for the course WMCS021-15 of
track Software Engineering and Distributed Systems
at University of Groningen under the supervision of
Dr. Fadi Mohsen (Computer Science, University of Groningen)
and
Dr. Michael Wilkinson (Computer Science, University of Groningen)
and
Thomas Rooijakkers (Cyber Security Researcher, TNO)
and
Thijs Klooster (Cyber Security Researcher, TNO)

Andrei-Claudiu Veres (s4047346)

October 10, 2023

Contents

Abstract

Web applications are foundational in today's digital landscape, necessitating advanced security measures. This study delves into Interactive Application Security Testing (IAST) and Web Fuzzing, two pivotal techniques for detecting web vulnerabilities. We systematically evaluate their strengths and weaknesses, emphasizing their potential in addressing vulnerabilities highlighted by the OWASP Top 10. While IAST excels in real-time vulnerability detection, Web Fuzzing offers an expansive approach, adept at uncovering elusive edge cases. Our research suggests that combining these techniques could lead to substantial enhancements in web application security. Additionally, we introduce the idea of an open-source IAST tool and contemplate the benefits that recent advances in artificial intelligence might bring to these techniques. Furthermore, we underscore the significance of understanding these tools' operation within the realm of cloud computing.

1 Introduction

A web application is a software application that is accessed via a web browser over the internet. Web technology has evolved from static and informative content to powerful, dynamic, and engaging materials [1]. In recent years, as more and more businesses are using the internet to offer their products and services, web-based applications have become the backbone of our modern digital society. The development lifecycle of these applications consists of multiple sequential stages. These are feasibility, analysis, design, coding, testing, and finally, implementation and maintenance [2]. These stages ensure not only the successful deployment of the application but also its correct functionality over time.

However, the increased use of web applications by businesses has brought unwanted attention from malicious actors. Now more than ever, the safety of web applications has become a great concern. Cybersecurity statistics from 2023 indicate that the number of cyber attacks per year is currently around 800,000. With the cost of a data breach averaging at $9.44M, it is predicted that cybercrime will cost $8 trillion by the end of 2023 [3]. A web application vulnerability is a security flaw or weakness that can be exploited by a malicious hacker to gain unauthorized access to a system's resources. The increasing frequency of these issues has escalated the demand for robust software security measures.

One entity that plays an important role in addressing these vulnerabilities is *The Open Web Application Security Project* (OWASP). This community-led, non-profit organization has tens of thousands of members. Its mission is to improve software security through open-source software projects and comprehensive resources that aid in the identification, prevention, and mitigation of web application vulnerabilities [4].

Software security has become an essential effort in order to handle web-based vulnerabilities in an efficient manner. This effort requires the implementation of security testing in the software development cycle of these applications. The aim is to prevent, detect, and rectify security vulnerabilities rapidly and efficiently. In this paper, we focus on *Interactive Application Security Testing* (IAST) and fuzzing. These are popular security techniques that are key in the identification of web application vulnerabilities.

The objective of this paper is to rigorously evaluate current techniques and tools used in OWASP vulnerability detection. We assess the strengths, weaknesses, and effectiveness of IAST and fuzzing in this context. These technologies were chosen as primary focuses due to their potential for enhancing software security in web-based applications. Furthermore, we are identifying potential gaps and areas for improvement in the vulnerability detection landscape with respect to the aforementioned technologies. The research helps the security community to better understand and choose the right software security testing technique and tool based on their needs.

We answer the following research questions:

- Q1: What are the current techniques and tools used in OWASP vulnerability detection, with a particular focus on IAST and fuzzing?
- Q2: What are the strengths and weaknesses of these techniques, and how effective are they in detecting OWASP vulnerabilities?
- Q3: What are the potential gaps or areas for improvement in current OWASP vulnerability detection techniques?

The paper is structured as follows: In Section 2, we begin with a *Literature Review*. Here, we discuss the background information and the context needed for understanding the thesis. Next, in Section 3, we discuss the *Methodology*, outlining our research approach. This is followed by Section 4, or *IAST Technique Analysis* (Q1 & Q2), and Section 5, *Fuzzing Technique Review* (Q1 & Q2), where we explore the main topics in more detail. Following that, we have Section 6, where we discuss the *IAST and Fuzzing Comparison and Future Directions* (Q3). We conclude with Section 7, *Conclusion*, and Section 8, *Limitations and Future Work*.

2 Literature Review

In this section, we cover essential topics that lay the groundwork for subsequent sections of the paper. We begin with an overview of the recent OWASP Top Ten report, followed by a discussion on web vulnerability detection methods, including white, black, and grey box approaches. We also introduce SAST and DAST, as they play a crucial role in one of the techniques detailed later in the paper. Lastly, we outline fundamental metrics used to assess the effectiveness of vulnerability detection.

2.1 OWASP vulnerability classification

There are many types of web-based application vulnerabilities. To defend against potential attacks, we first need to have a deep understanding of these vulnerabilities. On a high level, we can classify web vulnerabilities into three categories: **input validation** vulnerabilities (IPV), **session management** vulnerabilities (SMV), and **application logic** vulnerabilities (ALV) [5]. *OWASP Top Ten* is an awareness document for developers and security experts, created with the aim of representing the most critical and prevalent security risks for web applications [4]. In Figure 1 [4], the most recent classification of the most threatening risks associated with web applications is displayed.

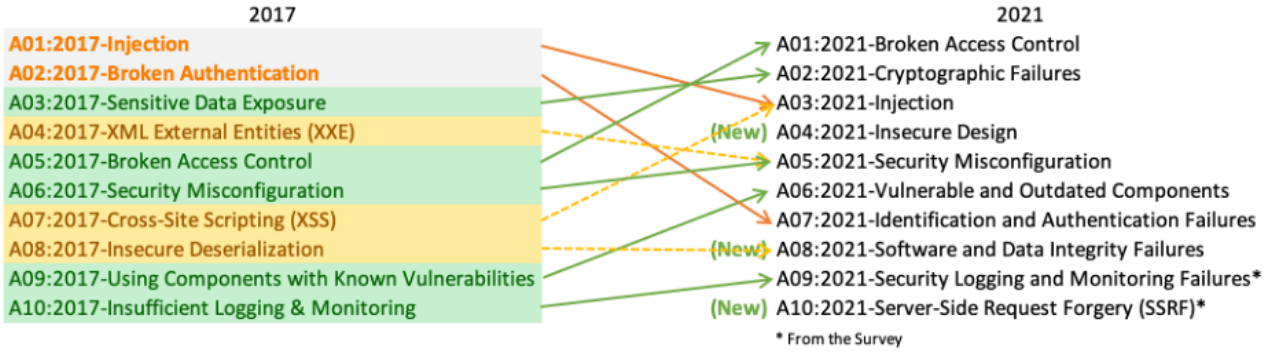

Figure 1: OWASP Top Ten Classification [4]

The color scheme in the above image represents the changes in the status of vulnerabilities ranked in the OWASP top 10 list. The *green* arrows signify vulnerabilities that have increased in importance and severity compared to 2017. The *orange* arrows indicate a decrease in the importance and severity of the ranked vulnerabilities. Meanwhile, the *yellow* arrows denote vulnerabilities that have been removed from the list and merged into other categories. The following list is based on the most recent OWASP vulnerability classification, year 2021, found on the right side of Figure 1:

1. **Broken Access Control** refers to the failure of implementing restrictions and policies on which actors can access specific resources. As the policy is written by humans, it is prone to error, therefore common and, unfortunately, critical in nature [6]. Some examples of these types of attacks are *Local File Inclusion Attack* and *Remote File Inclusion Attack*.

2. **Cryptographic Failures** indicate successful decryption of maliciously gathered data. For example, the use of old or outdated ciphers may lead to quick and easy data decryption by the hacker [6]. These types of attacks lead to sensitive data exposure. The most common attacks of this type are: *Information Leakage Attack* and *Database Theft* [7].

3. **Injections** are cyber-attacks where the attacker sends malicious data as part of a command or query. The aim of these attacks is to trick the interpreter into executing these commands,

thereby illicitly retrieving, modifying, or deleting data. The most critical injection attacks are: *SQL Injections*, *Code Injections*, and *XPATH Injections* [7]. As indicated by the OWASP top 10 [4], since 2021, *Cross-Site Scripting (XSS)* has also been classified as an injection attack.

4. **Insecure Design** occurs when a web application allows direct access to objects based on the input supplied by the user. In this case, an attacker may be able to bypass authorization and directly access system resources. Several types of attacks can be conducted by exploiting this type of vulnerability: *Path Traversal Attack*, *Direct Request Attack*, and *Authorization Bypass Through User-Controlled SQL Primary Key* [7].

5. **Security Misconfiguration** is an issue that arises due to the misconfiguration of one or more components of the system. Secure settings must be defined, implemented, and maintained. Unfortunately, in many cases, these components remain with their default settings. A plethora of attacks can occur due to misconfiguration, and their severity depends on the level and location of the misconfiguration [7]. Some examples of misconfiguration vulnerabilities are: *unimplemented HTTP security headers*, *unprotected files and directories*, *poorly configured permissions*, *error messages that reveal sensitive information about the system*, *enabled default accounts and passwords*, and *enabled unnecessary exploitable features* [6].

6. **Vulnerable and Outdated Components** are software components that contain already known security issues or are no longer supported by their developers and, therefore, are prone to security breaches. These components could be libraries, frameworks, APIs, or any other such elements used in the creation of the web application. The severity of the vulnerability depends on the nature of the component and the level at which it is utilized [6] [7].

7. **Identification and Authentication Failure** refers to the practice of exploiting vulnerabilities during the authentication process in a web application and the malicious use of user credentials to gain unauthorized access to information. These attacks can be classified into the following categories: *Brute Force Attack*, *Dictionary Attack*, *Credential Enumeration Attack*, *Session Fixation Attack*, and *Cookie Poisoning Attack* [7].

8. **Software and Data Integrity Failures** occur during the software development cycle due to the use of modules, libraries, and plugins from untrusted third-party sources. If these components are not properly maintained and validated, they may lead to vulnerabilities. Some scenarios where the system is susceptible to compromise are: *missing platform patches*, *missing unit tests*, *incomplete vulnerability scanning*, and *improper input validation* [6].

9. **Security Logging and Monitoring Failures** refer to the insufficient logging and monitoring of a system's activity. Logging is the process of recording the activity of a system. When a system is not properly recording and monitoring its activity, detecting security breaches becomes more difficult [6]. This vulnerability risk is due to a lack of a logging mechanism, failure to log relevant information, not storing the logs for a sufficient amount of time, or an ineffective monitoring process.

10. **Server-Side Request Forgery (SSRF)** is a vulnerability risk that can be exploited by an attacker to make a server send an unintended request. It results from inadequate validation and restrictions on the side of the web application. This type of vulnerability enables the attacker to perform malicious actions such as *scanning internal networks* and *exploiting unsecured services* [6].

2.2 White Box, Black Box and Grey Box

While there are numerous security testing techniques, they can be categorized into three groups. These are *white box testing*, *black box testing*, and *grey box testing*. In this section, we briefly explain each category and discuss the advantages and disadvantages that come with their implementation.

White box testing, also known as glass box testing or open box testing, refers to a group of techniques that utilize the internal knowledge of a system to perform testing. In this type of testing, the security specialist is aware of the system's architecture, has access to the source code, and understands the underlying technology. Various techniques are used, but the overall process generally involves the following steps: the testers use their system knowledge to develop specific testing scenarios, they create the required data for these scenarios, then they run these tests, and finally, they analyze the results. Some of the advantages that come with white box testing are: the revealing of errors in the code and maximum coverage attained during testing. Unfortunately, this approach has its disadvantages: it is time-consuming, requires a skilled tester, risk of testing bias, some paths may remain unchecked, and some code might be omitted [8].

At the opposite end of the spectrum, we find **black box testing**. The techniques that fall under this category treat the software as a "Black Box," implying that testing is performed without any knowledge of the system's internal workings. Although the methodologies applied in black box testing have similarities with those in white box testing—particularly in evaluation procedures—the distinguishing characteristic is the tester's lack of architectural knowledge about the system and inability to access its source code. The advantages of using this kind of approach are: efficiency for large code segments, simple tester understanding, decreased of testing bias and rapid test case development. However, the implementation of black box testing comes with some drawbacks: limited coverage, difficulty in designing tests, and potentially inefficient testing [8].

Grey box testing aims to get the best from white box testing and mix it with the best of black box testing. This hybrid approach allows the tester to have some knowledge about the internal structure of the system and design efficient target testing accordingly. This approach has increased coverage compared with black box testing while taking less time to execute than white box testing. Some of the advantages of grey box testing are: enhanced test scenarios, employing the user's point of view, and unbiased testing. However, there are some disadvantages such as: limited test coverage due to inaccessible source code, paths that may remain untested, and the potential to re-run test cases (redundancy) [8]. Nevertheless, grey box testing has great potential when it comes to implementing security in the development cycle of a web application. In this paper, we discuss interactive application security testing (IAST) section 4, which is an interesting grey box testing approach.

2.3 Static and Dynamic Application Security Testing

Static Application Security Testing (SAST) is a methodology used in the identification of application security vulnerabilities. It is a type of white-box security testing. The tester has access to the source code and is familiar with the architecture of the application, network, and all other necessary information regarding the system. The strength of SAST is the ability to pinpoint the vulnerabilities inside the source code. Additionally, SAST checks for code coverage, data flow testing, path testing, and loop testing [9].

However, SAST does not evaluate runtime data flow and operational security gaps. **Dynamic Application Security Testing (DAST)** addresses this limitation by implementing a black-box-like ap-

proach. The tester has no knowledge about the structure of the system, nor does he have access to the source code. He uses the exposed (user) interface to perform the tests. The idea behind DAST is to attack the application from the point of view of a malicious actor. The aim is to identify runtime vulnerabilities such as I/O validation issues, server configuration mistakes, and other such weaknesses [9].

Unfortunately, DAST often fails to indicate precisely where in the source code the vulnerability is located. **Interactive Application Security Testing (IAST)** is a mixture of the DAST and SAST approaches. In this paper, we discuss how SAST and DAST can be combined to obtain enhanced security performance.

2.4 Vulnerability Detection Efficiency Metrics

One of the most important metrics in the measurement of vulnerability detection is *accuracy*. To understand this metric, it is essential to grasp the concepts of true positives, false positives, true negatives, and false negatives in the context of vulnerability detection [10].

- **True Positives (TP)**: In this scenario, the security approach correctly identifies a vulnerability. The vulnerability exists and the technique correctly flags it.
- **False Positives (FP)**: In this case, the security method incorrectly identifies a vulnerability. The technique flags a vulnerability, but in reality, there is no vulnerability—it's a false alarm.
- **True Negatives (TN)**: In this situation, the security approach correctly identifies that a vulnerability does not exist. No vulnerability is flagged.
- **False Negatives (FN)**: Under these circumstances, the security method fails to identify a vulnerability. Although the vulnerability exists, the technique fails to flag it.

Accuracy can be calculated using the following formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

3 Methodology

In this section, we discuss the methodology used to answer our research questions.

3.1 Methodology for IAST

The methodology for this section is focused on the analysis of IAST techniques, especially in relation to OWASP classified vulnerabilities. This analysis is divided into three main parts: the current state of IAST, case studies on IAST, and a comparison of IAST tools.

The first part looks at the current state of IAST techniques. A literature review was carried out using search queries in Google Scholar with terms such as “IAST”, “SAST”, “DAST”, “interactive application security testing”, “static application security testing”, “dynamic application security testing”, “state of the art”, “OWASP”, and “OWASP Top 10”. Initially, 15 papers were chosen based on their titles, but this was narrowed down to seven after reviewing the abstracts to assess their relevance. This part explores the historical background of IAST, the pros and cons of SAST and DAST approaches, and how these relate to IAST. It also explores the techniques used in IAST, OWASP Top 10 vulnerabilities coverage, and how effective some tools are at identifying these vulnerabilities.

The second part discusses two case studies on IAST to understand what an IAST implementation can look like and the possible improvements it can bring. The first case discusses a completely open-sourced IAST implementation, while the second case examines the improvements achieved by a DAST tool that uses information from inside the application.

The final part focuses on a discussion of two IAST tools: Seeker by Synopsys [11] and Contrast Assess by Contrast Security [12]. This choice was made considering various factors, including the absence of mainstream open-sourced IAST tools. Therefore, the discussion revolves around only closed-sourced IAST tools. Seeker was chosen because it is a pioneering tool in the IAST domain, and Contrast Assess was selected for its high user satisfaction ratings.

In assessing these tools, different aspects are discussed, including: type of testing (black/white/grey box), main components of the tool, integration with development environments, detection techniques, OWASP Top 10 classified vulnerabilities coverage, real-time analysis and feedback, reporting and dashboard features, pricing and licensing, community and vendor support, and support for different programming languages and platforms. This discussion aims to give an overview of the tools rather than an exhaustive analysis.

3.2 Methodology for Fuzzing

The methodology for this section is centered around the analysis of fuzzing techniques, specifically web fuzzing. This analysis is divided into three main parts, each focusing on a different aspect of fuzzing techniques.

The initial segment offers a perspective on the current state of fuzzing techniques. A literature review was undertaken, employing search queries in Google Scholar using the following terms: “fuzzing,” “state of the art,” “black-box,” “white-box,” and “grey-box”. Initially, 25 papers were chosen based on their titles, but this was narrowed down to 13 after reviewing the abstracts to assess their relevance. T In addition, a study from TNO’s prior research was included [13]. This part discusses the historical context of fuzzing, the functioning and components of a fuzzer, and the relevant definitions in the field of fuzzing. It also delves into the distinctions between black, white, and grey box fuzzing.

The subsequent segment is dedicated to web fuzzing. It highlights how web fuzzing stands apart from other fuzzing genres and investigates which vulnerabilities, as listed in the OWASP top 10, can be pinpointed using web fuzzing. The data for this part was gathered following a similar literature review approach as the prior segment, incorporating additional search terms like “web application” and “OWASP”.

The concluding segment dives into an analysis of two web fuzzers: Wfuzz [14] and SQLMap [15]. The rationale behind their selection is based on several criteria:

- The fuzzer should target at least one vulnerability listed in OWASP.
- To avoid obsolete tools, the fuzzer must have been updated at least once in the past five years.
- The fuzzer should be well-documented.
- Preferably, the tool should primarily function as a fuzzer rather than a multifaceted instrument like a web application vulnerability scanner.
- For better understanding the inclusion of both a multi-test fuzzer and a specific fuzzer is required.

In the assessment of these fuzzers we discuss aspects such as the type of testing (black/white/grey box), the nature of the fuzzer (multi-test/specific test), the fuzzing technique employed, proficiency in detecting web vulnerabilities, relevant features, pricing, customizability, community and vendor support, and supported platforms. These criteria aim to offer an overview of the possibilities rather than an exhaustive analysis.

4 IAST Technique Analysis

In this section, we start by exploring the state of the art in IAST and its integration of SAST and DAST, particularly in relation to the OWASP Top Ten. We then examine two case studies, focusing on efforts to develop open-source IAST frameworks. To conclude, we evaluate two IAST tools, comparing them based on various criteria.

4.1 State of the Art of IAST

In recent years, there has been an increasing need in the cybersecurity field for robust and comprehensive testing mechanisms capable of automating, at least partially, the evaluation of security aspects in software applications. A notably efficient solution in this endeavor, especially for web applications, is Interactive Application Security Testing (IAST). Initially proposed and developed by Synopsys [11] [16], IAST leverages two well-established techniques: Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) [17].

As discussed in subsection 2.3, SAST operates as a white-box testing method, whereas DAST functions as a black-box testing method. Employing SAST in security testing of applications provides numerous benefits, including the identification of complex vulnerabilities in the early stages of development. Since it pinpoints issues directly in the source code, it facilitates convenient resolution for developers. Additionally, it integrates seamlessly into the development life cycle and, on average, identifies more vulnerabilities than DAST [18] [19]. However, this approach does come with drawbacks such as a reliance on the specific programming language and framework used, a tendency for false positives or negatives, and potential difficulties in reproducing identified vulnerabilities [18] [17] [20].

Utilizing DAST, on the other hand, enables developers to uncover vulnerabilities at runtime that are undetectable by SAST, presents fewer false positives, and offers both a cost-effective solution and independence from the application's programming language [18] [20]. Nevertheless, it also holds disadvantages such as its confinement to the later stages of the development cycle, limitations imposed by the tool's vulnerability database, lack of feedback on the vulnerability detected, and a prolonged testing duration for web applications [18].

IAST emerges as an intelligent combination of SAST and DAST, significantly enhancing the accuracy of vulnerability detection in web applications [16] [18] [19] [20] [17]. By providing real-time code analysis while operating directly on the server integrated into the web application, IAST can promptly identify vulnerable source code during the early development stages. This is not only convenient for developers but also saves financial and computational resources. Additionally, this approach lowers the risk of exploitation of vulnerabilities and lower development costs to fix the issue. Its implementation as a server agent allows it to gather information on application behavior generated by end-users, sanitize user input to prevent injection attacks or remote executions, and identify vulnerabilities with increased accuracy. However, IAST has its shortcomings, including language dependency, inability to detect client-side vulnerabilities, and the potential to incur runtime overhead on the application server.

One of the primary techniques utilized in implementing the IAST approach is *taint analysis*. This method is leveraged through dynamic taint analysis in IAST to monitor the flow of information between sources and sinks during runtime [20]. In this context, a program value deemed as "tainted" is the result of computations based on data derived from a taint source, while all other values are con-

sidered "untainted." Another pivotal technique in IAST implementations is *symbolic execution*. This technique involves formulating a logical formula that represents a program's execution, facilitating the analysis of the program's behavior across a wide array of inputs simultaneously [20].

In Table 1, we highlight the OWASP Top 10 classified vulnerabilities that can be identified through an IAST approach. This assessment is partially based on previous research that explored the coverage for OWASP TOP 10 2017 [18] and a more recent study on OWASP TOP 10 2019 [19]. IAST tools excel in monitoring and analyzing the runtime behavior of applications, including the enforcement of access controls. By scrutinizing HTTP requests and responses, these tools can pinpoint instances where access controls are not enforced properly, therefore identifying *broken access control* vulnerabilities.

Furthermore, an IAST approach can detect *injection* vulnerabilities, which frequently arise due to improper handling of inputs. This is achieved by monitoring the application during runtime and identifying points where untrusted inputs are used. Similarly, *security misconfigurations* can be identified by examining the application's behavior under various settings. Testing different configurations might uncover issues like improper error handling or insecure default settings, indicative of misconfigurations. The monitoring capability of IAST extends to the authentication process, offering an opportunity to reveal vulnerabilities associated with *identification and authentication failures*. This process relies on a careful analysis of user authentication handling at runtime.

Moreover, IAST can identify *software and data integrity failures* by overseeing how the application manages data. For instance, it can identify insecure deserialization incidents, potentially leading to data integrity failures. Lastly, the technique proves efficient in recognizing *server-side request forgery (SSRF)* vulnerabilities by supervising the application's handling of external resource requests. When the behavior showcases signs of an SSRF attack, such as unexpected outbound requests, the tool raises an alarm to signal a potential attack.

The detectability of a couple of OWASP classified vulnerabilities, using an IAST approach, remains a subject of debate. In regard to *cryptographic failures*, IAST could potentially spot certain irregularities, such as the transmission of sensitive data in unencrypted form. Although IAST tools may not be initially designed to find *vulnerable and outdated components*, they might still be capable of identifying anomalies in observable runtime behaviors. Developers can utilize IAST tools to help identify security issues that need to be logged and monitored, and to verify whether the actual monitoring systems is correctly flagging these security concerns, essentially serving as a form of double-check. In this respect, it could be contended that an IAST approach may provide indirect assistance in recognizing *security logging and monitoring failures*.

However, it should be noted that there are vulnerabilities in the OWASP Top 10 list that are arguably outside the identification purview of IAST approaches. For instance, the *insecure design* flaw, a high-level security issue, is generally identified by software architects through a detailed analysis of the application's structure. Given that IAST primarily functions during runtime, it is not designed to pinpoint vulnerabilities associated with software architecture design.

There are a couple of studies that delve into the effectiveness, efficiency, and accuracy of IAST tools [20] [19]. We summarize the key findings from these studies regarding IAST as follows:

- IAST can detect a wider range of vulnerabilities compared to either SAST or DAST when used individually.
- The IAST approach is capable of identifying a larger variety of vulnerability types in comparison to SAST and DAST.

- Utilizing a synergized approach that combines multiple tools yields better performance in detecting vulnerabilities while minimizing false positives (two IAST tools yields the best results).
- Employing IAST can lead to a significant reduction in the number of false positives.

Table 1: OWASP Top Ten Vulnerabilities and IAST Identifiability

OWASP Vulnerability	**Identifiable by IAST**
A01:2021-Broken Access Control	✓
A02:2021-Cryptographic Failures	∼
A03:2021-Injection	✓
A04:2021-Insecure Design	×
A05:2021-Security Misconfiguration	✓
A06:2021-Vulnerable and Outdated Components	∼
A07:2021-Identification and Authentication Failures	✓
A08:2021-Software and Data Integrity Failures	✓
A09:2021-Security Logging and Monitoring Failures	∼
A10:2021-Server-Side Request Forgery	✓

4.2 Case Studies on IAST

A recent case study explores the vulnerabilities present in an X-Government web-based application, leveraging the Interactive Application Security Testing (IAST) methodology [18]. The authors integrate Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) processes, facilitating both pre-run and runtime analyses. The outlined method goes through several steps including target discovery, vulnerability scanning, and results analysis. This project leverages well-known open-source tools to embody the IAST strategy, employing Jenkins [21] for integration, ZAP [22] for DAST, and SonarQube [23] for SAST. The methodology was tested against the OWASP Benchmark [24], hence ensuring a detailed vulnerability assessment.

Following this methodology, the study discovered 249 vulnerability risks, covering issues like injections, broken authentication, and security misconfigurations. When compared with standalone SAST or DAST approaches, the IAST strategy outperforms in finding a more extensive range of vulnerabilities. As of the time of this report, to the best of our knowledge, this specific implementation of the IAST approach is the only fully open-source IAST tool available. This case study supports the use of the IAST approach, and its significantly promising role in security testing.

In another case study, the effectiveness of DAST was enhanced by leveraging information about the web application gathered using SAST [17]. The authors highlight that in recent years, the potential of DAST has declined due to the restrictive filters introduced in web applications. These filters limit the input parameters, consequently diminishing the capacity to identify vulnerabilities. To counter this, they propose a solution formulated around an IAST approach. During static testing, vital details such as the URL, parameter names, and filter constraints are stored. Subsequently, before initiating dynamic testing, attack strings that can bypass the established filters are requested to facilitate a more robust vulnerability detection process.

In the practical application of this concept, the authors leveraged well-established open-source software applications to implement and validate their strategy. The experiment utilized SPARROW [25] for SAST, ZAP [22] for DAST, and Z3str2 [26] for string constraint resolution. They employed the OWASP Benchmark [24] to assess the approach, focusing strictly on cross-site scripting vulnerabilities. The results are encouraging, demonstrating a significant improvement with a 33% increase in the vulnerability detection rate when applying the IAST methodology as opposed to solely applying DAST.

4.3 IAST Tools Comparison

4.3.1 Seeker

Seeker, developed by the Synopsys company, is a reputable interactive application security testing tool [11]. It employs grey box testing methodologies to facilitate in-depth analysis of web applications, offering real-time feedback encouraging secure coding habits, and it can be easily integrated into various development environments.

1. **Type of Testing**: Seeker functions as a grey-box testing tool, delving into the application's internals in real-time to identify vulnerabilities.

2. **Components**: Seeker has two main components. The *Seeker Server* provides the user interface, collects vulnerabilities and stores the vulnerabilities generated by the agent. Meanwhile, the *Seeker Agent* is tasked with detecting vulnerabilities within the application, diligently monitoring the web application server's data flow from the initial request to the final response from the application.

3. **Integration with Development Environments**: Seeker seamlessly integrates with many CI/CD pipelines, including Jenkins [21] and GitLab [27], and is compatible with IDEs such as Visual Studio [28] and Eclipse [29]. It is designed for containerized and cloud environments like AWS [30], Azure [31], and Google Cloud [32]. Moreover, it facilitates continuous feedback in Agile and DevOps workflows.

4. **Detection Techniques and Methodologies**: Seeker utilizes *active verification technologies* to identify potential vulnerabilities and confirm their actuality and exploitability. It *tracks sensitive data* through the data flow to scrutinize the security of data transmission and storage. Moreover, it ensures alignment with OWASP standards and compliance with PCI DSS and GDPR regulations.

5. **Detected Web Vulnerabilities (OWASP Top 10)**: Seeker is proficient in detecting a vast array of web vulnerabilities, prominently those listed in the OWASP Top 10. Its capabilities align with the details discussed previously and presented in Table 1.

6. **Real-time Analysis and Feedback**: Seeker provides real-time feedback coupled with contextual assistance to developers, pinpointing potential security issues and offering remediation suggestions.

7. **Reporting and Dashboard Features**: The dashboard offers an extensive overview of the application's security status, detailing the vulnerabilities identified, their severity, and the progression of remediation efforts, enabling developers to garner profound insights into the security landscape.

8. **Pricing and Licensing**: While the official website lacks pricing details, secondary sources indicate a user-based licensing system with an annual fee of 70,000 dollars for 50 users [33].

9. **Community and Vendor Support**: Despite the restricted information available without product registration, Seeker maintains regular updates, with the most recent being less than a month prior to this paper's drafting [34]. Synopsys fosters a community forum and supplies extensive documentation to aid users.

10. **Supported Programming Languages and Platforms**: Seeker is compatible with Windows, Linux, and macOS, supporting a range of technologies including Java, DotNet, PHP, C#, JavaScript, PLSQL, TSQL, Groovy, Scala, and Clojure [35].

4.3.2 Contrast Assess

Contrast Assess is an interactive tool for testing the security of applications, created by the Contrast Security Company [12]. It uses grey-box testing, which combines DAST and SAST methods, to create a final product that can be easily integrated at various stages of the software development life cycle. The tool supports many programming languages and gives real-time feedback along with detailed reports, helping developers to build more secure web applications.

1. **Type of Testing**: Since Contrast Assess utilizes an IAST approach, it falls under the category of grey-box testing tools.

2. **Components**: Contrast Assess comprises two primary components. The *Contrast Platform* serves as the central hub where all vulnerability data and analytics are stored, offering deeper insights into the security status of the web application. Meanwhile, the *Contrast Agents* are embedded in the application's runtime environment, continuously monitoring the behavior and data flow to identify potential vulnerabilities in real time.

3. **Integration with Development Environments**: Contrast Assess is designed to seamlessly integrate with existing CI/CD pipelines, including Jenkins [21]. It also offers integration with IDEs such as Visual Studio [28] and IntelliJ IDEA [36].

4. **Detection Techniques and Methodologies**: The tool offers more than just code analysis; it brings insights from the runtime environment as well. One notable feature is *path mapping*, which intelligently maps out all the routes within an application, helping to identify vulnerabilities that might be exploited through specific paths.

5. **Detected Web Vulnerabilities (OWASP Top 10)**: Contrast Assess is designed to discover vulnerabilities classified under the OWASP Top 10. The tool provides coverage as indicated in Table 1.

6. **Real-time Analysis and Feedback**: Contrast Assess offers instant feedback to developers, along with suggestions for fixing identified security issues. It also provides educational resources to help developers better understand and address the vulnerabilities highlighted.

7. **Reporting and Dashboard Features**: Contrast Assess offers detailed reporting, including an assessment of vulnerability severity, accessible via a centralized dashboard. This dashboard presents a comprehensive view of the application's security status, facilitating easy monitoring and management.

8. **Pricing and Licensing**: Pricing details are available upon request from the Contrast team. Based on information from various websites, it appears that the pricing is user-based; for instance, a package for 10 developers is priced at around 28,000 dollars per year [37].
9. **Community and Vendor Support**: Contrast Assess maintains a community forum where users can engage with other professionals on a variety of topics related to application security. The vendor provides robust support, including monthly updates; the most recent update was released approximately one month prior to the writing of this paper.
10. **Supported Programming Languages and Platforms**: Contrast Assess is compatible with Windows, Linux, and Mac OS, and supports a wide array of technologies including Java, JavaScript, .NET, .NET Core, Node.js, Ruby, Python, Scala, PHP, Kotlin, Vue.js, and TypeScript [38].

5 Fuzzing Techniques Review

In this section, we start by examining the state of the art in fuzzing techniques, focusing particularly on web fuzzing. We delve into the mechanics of a web fuzzer, especially in relation to the OWASP Top Ten. To conclude, we assess two fuzzers, comparing them based on various criteria.

5.1 State of the Art of Fuzzing

Introduced by Prof. Barton Miller in 1989, *fuzzing* is a methodology where a program is run repeatedly with input data that is either artificially or semantically altered [39]. This technique is frequently utilized to detect security-related bugs within programs. *Fuzz testing* involves the application of fuzzing techniques to a program with the objective of examining whether it adheres to its correctness policy [40].

A *fuzzer* is a specific program designed to conduct fuzz testing on other programs [40]. In the context of software security, fuzzers serve as valuable tools for security specialists, enabling them to analyze various facets of a program. These tools prove particularly effective in identifying potential issues across a range of software types, encompassing compilers and interpreters, application software, network protocols, and operating systems [41].

Research discusses the general fuzzing process [41], as shown in Figure 2. The *target program* is the program under test. The *monitor* component, mainly used in white-box and grey-box fuzzing, collects relevant runtime information about the program, and is not required for black-box fuzzers.

The *test case generator* has two main implementation methods: mutation-based and grammar-based. In the mutation-based method, seed files serve as starting points; they are original inputs that are modified to produce new test cases. This method generates test cases by altering these seed files either randomly or using predefined strategies. Meanwhile, the grammar-based method employs predefined rules based on the target's specification. The *bug detector* is a module designed to collect and analyze crashes or errors reported during fuzzing, and anything flagged by it is sent to the final component, the *bug filter*. In this final step, bugs are sorted and filtered for the convenience of security specialists.

Depending on the amount of information required from the targeted program by the fuzzing technique, we can categorize it into: black-box fuzzing, white-box fuzzing, and grey-box fuzzing [41][40][13].

Black-box fuzzing does not necessitate any information from the program under test. It generates random input from scratch or by mutating a provided seed using predefined rules. However, due to its "blind" approach, black-box fuzzing suffers from low test coverage.

To address the low coverage issue of black-box fuzzing, researchers developed *white-box fuzzing*. This technique gathers knowledge about the internal logic of the target program. Despite enabling the security specialist to evaluate nearly all execution paths, it also brings considerable overhead and implementation difficulties compared to its counterpart.

Finally, *grey-box fuzzing* stands between the previously mentioned methods. The fuzzer has some system knowledge in this case. By analyzing the target program at runtime and mutating the input data accordingly, the fuzzer obtains improved code coverage. This approach enhances code coverage compared with black-box fuzzing, while maintaining lower overhead compared to white-box fuzzing.

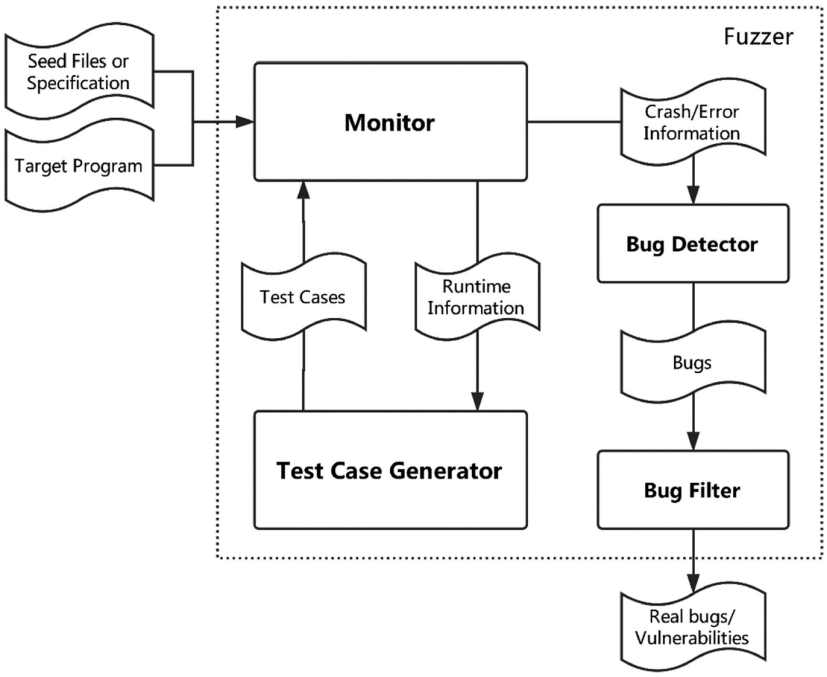

Figure 2: General process of fuzzing [41]

5.2 Web Fuzzing

Web fuzzing refers to the action of fuzzing a web application with the goal of finding bugs or security flaws. The most important security concerns found in web applications are classified in OWASP Top 10 [4]. Web fuzzing is achieved by sending a set of HTTP requests to the program under test and analyzing its behavior when receiving various inputs [42]. In short, the tester has to configure the input that will be generated and after the HTTP request has been sent they analyze whether any vulnerabilities has been found, consequently creating a testing report [42].

Although some research indicate a lack of research of web fuzzing [13][43], especially compared with other use cases of fuzzing, recent work indicate that web fuzzing is an effective way to detect web vulnerabilities [44].

Input generation mechanism is an essential part of a web fuzzer. The generated input applies to a part of the HTTP request sent to the application under test. The input generation can be achieved by using *fuzz vectors*, *number generator*, *brute force* or similar string generator [42]. Furthermore, the generated input can be further modified with *string substitution*, *case modification* and *encoding* [42].

Vulnerability detection poses a challange as fuzzers, by their nature, send a huge number of HTTP requests to test the web application. In order to filter the responses, the web fuzzers provide features that help the tester identify vulnerabilities: *HTTP response error*, *timeout* or *specific content* (of the HTTP response) [42]. Based on the number of types of vulnerabilities that the fuzzer is capable of testing, we can classify them as *multi-test fuzzers* and *specific vulnerability fuzzing* [45].

In Table 2, we highlight which OWASP Top 10 vulnerabilities [4] can be identified using fuzzing techniques. Prior research has shown significant results in vulnerability identification through *injection* [43] [42] [45] [46]. Vulnerabilities are identified by sending malicious input to the application,

including but not limited to SQL injections, command injections, and cross-site scripting (XSS). A recent paper demonstrated how fuzzing can be used to detect *Security Misconfigurations* [47]. It's worth noting that while the approach described in this study incorporates a web crawler component to pinpoint input points, it primarily relies on fuzzing to generate random input that is subsequently sent to those input points.

Software and Data Integrity Failures can be detected using fuzzing [48]. For example, fuzzing is effective in identifying insecure deserialization issues. Web applications that do not rigorously verify the integrity of serialized data are vulnerable to such threats. Furthermore, fuzzing techniques can also be employed to detect *Server-Side Request Forgery* [49]. Sending various payloads to the server may prompt the server to initiate unintended requests.

While the rest of the OWASP Top 10 vulnerabilities continue to pose significant concerns, this research posits that it is improbable for fuzzing techniques to successfully identify these vulnerabilities. *Broken Access Control* and *Identification and Authentication Failures* are threats that can be discerned either through manual testing or with software specifically tailored for this purpose. *Cryptographic Failures* necessitate a profound grasp of cryptographic implementations and are not straightforwardly identifiable using fuzzing.

Insecure Design stands out as a high-level vulnerability. To leverage it, an in-depth review of the system's architectural design is requisite. The challenges associated with *Vulnerable and Outdated Components* are best addressed using dependency scanning tools. Hence, there exists no reason to resort to fuzzing in this context. Finally, *Security Logging and Monitoring Failures* concern the lack of a proper logging/monitoring system, which is not related to fuzzing.

Table 2: OWASP Top Ten Vulnerabilities and Fuzzing Identifiability

OWASP Vulnerability	**Identifiable by Fuzzing**
A01:2021-Broken Access Control	×
A02:2021-Cryptographic Failures	×
A03:2021-Injection	✓
A04:2021-Insecure Design	×
A05:2021-Security Misconfiguration	✓
A06:2021-Vulnerable and Outdated Components	×
A07:2021-Identification and Authentication Failures	×
A08:2021-Software and Data Integrity Failures	✓
A09:2021-Security Logging and Monitoring Failures	×
A10:2021-Server-Side Request Forgery	✓

One of the primary metrics for evaluating fuzzing techniques is accuracy. This pertains specifically to the precision of fuzzers, tools designed to implement these techniques. In certain scenarios, a fuzzer can be directly tested in its original form. However, in the context of web application vulnerability detection, fuzzers typically serve as integral components of a web application vulnerability scanner. This integration makes it difficult to accurately evaluate the accuracy of fuzzing against vulnerabilities as classified by OWASP. Such assessments are complicated by variables like the effectiveness of the crawling mechanism and the precise implementation of the fuzzing strategy.

We were unable to locate prior research that clearly indicates the general accuracy of fuzzing in relation to identifying vulnerabilities as per OWASP standards. Given the volume of academic papers on fuzzing web applications focused on *injection* vulnerabilities, it suggests a heightened interest in exploiting this type of vulnerability compared to others. Nevertheless, such a generalization remains a limitation for this paper.

5.3 Fuzzing Tools Comparison

5.3.1 Wfuzz

Wfuzz is an open-source tool designed for assessing the security of web applications [14] [50]. It aids penetration testers by fuzzing web application parameters, enabling them to identify potential vulnerabilities. While Wfuzz can be used as a standalone tool, it is often integrated within larger systems, like web application vulnerability scanners, to automate the process [51]. Below, we delve deeper into the specifications of this fuzzer:

1. **Type of Testing**: Wfuzz functions as a *black-box* testing tool, meaning it doesn't necessitate prior knowledge of the application's internals. Instead, it focuses on understanding the application's behavior by sending a variety of inputs and evaluating the subsequent responses.
2. **Nature of the Fuzzer**: Wfuzz is a *multi-test* fuzzer, purposefully crafted to uncover a range of vulnerabilities in web applications.
3. **Fuzzing Techniques**: The primary technique employed by Wfuzz is *mutation-based* fuzzing. It leverages predefined payloads that are modified to generate individual test cases.
4. **Detected Web Vulnerabilities**: While the exact detection capabilities hinge on various factors like configurations and the payloads used, Wfuzz can potentially identify vulnerabilities such as Injection Flaws, Authentication and Session Management issues, Directory Traversal, and other Misconfigurations [52] [53].
5. **Features**: Wfuzz has several notable features, including Subdomain Fuzzing, Directory Fuzzing, Cookie Fuzzing, Header Fuzzing, and HTTP OPTIONS fuzzing [53].
6. **Pricing**: As of the release of this paper, Wfuzz remains an open-source tool and is freely accessible to users.
7. **Customizability**: Wfuzz offers *high levels of customizability*. Users have the option to define or select from a vast array of payloads, utilize different encoders, and even combine payloads using iterators [54].
8. **Community and Vendor Support**: Despite Wfuzz's repository being dormant for the past three years, it retains its significance and is still a part of the default package set in Kali Linux [55].
9. **Supported Platforms**: Wfuzz, being written in Python, is *platform-agnostic*. This ensures its compatibility across all platforms that support Python, such as Windows, macOS, and Linux.

5.3.2 SQLMap

SQLMap is an open-source penetration testing tool designed to automate the detection and exploitation of SQL Injection flaws [56] [15]. Though it is more sophisticated than a fuzzer, one could argue

that its approach to assessing vulnerabilities incorporates fuzzing techniques. Below, we delve deeper into the specifications of this tool within the context of web applications:

1. **Type of Testing**: SQLMap primarily operates as a *black-box* testing tool. It doesn't require knowledge about the internal structure or workings of the web application. Instead, it discerns vulnerabilities based on the responses from the application.
2. **Nature of the Fuzzer**: SQLMap is a *specific fuzzer*, tailored to detect SQL injection vulnerabilities.
3. **Fuzzing Techniques**: Although SQLMap employs a mix of techniques, it predominantly leverages *mutation-based* fuzzing.
4. **Detected Web Vulnerabilities**: SQLMap is designed to pinpoint a specific type of vulnerability classified by OWASP. It's primarily used to detect injection vulnerabilities, especially SQL injections.
5. **Features**: SQLMap is multifaceted and boasts numerous features. Here are some of the most prominent ones relevant to web applications [57]:
   (a) Automatic detection of SQL injections.
   (b) Support for most prevalent databases.
   (c) Compatibility with various SQL injection techniques, including: boolean-based blind, time-based blind, error-based, UNION query, and stacked queries.
6. **Pricing**: As of the publication of this paper, SQLMap continues to be an open-source tool and is freely available for users.
7. **Customizability**: Having an open-source framework, SQLMap is *highly customizable*. Additionally, the tool offers an extensive set of command-line options to fine-tune testing.
8. **Community and Vendor Support**: The community steering its development is active. The most recent update, at the time of writing, is less than a week old. The tool is included by default in Kali Linux [58].
9. **Supported Platforms**: Since SQLMap is developed in Python, it's *cross-platform* and can run on any system supporting Python, such as Windows, macOS, and Linux.

6 IAST and Fuzzing Comparison and Future Directions

In this paper, we have explored two prominent methodologies within the continuously evolving realm of application security: Interactive Application Security Testing (IAST) and fuzzing. These methodologies have been examined with a specific focus on detecting vulnerabilities in web applications. While previous sections delved into each method individually, this section juxtaposes them. We aim to highlight their strengths and weaknesses, elucidate the potential synergies between them, and contemplate prospective future directions.

6.1 Strengths

Table 3: Strengths of IAST vs. Web Fuzzing

Methodology	**Strengths**
IAST	• Real-time vulnerability detection. • Precise location identification of vulnerabilities in source code. • Intelligent combination of SAST and DAST. • Low false-positive rates. • Provides contextual information about vulnerabilities. • Operates across multiple languages and frameworks. • Reduced false negatives due to runtime operation.
Web Fuzzing	• Exceptional at identifying edge-case vulnerabilities. • Automated, continuous testing. • Operates independently of the application's internal structure. • Detects runtime issues like memory leaks and crashes. • Easy to replicate attack as the HTTP request is provided. • Versatile across various software types. • Discovers previously unknown vulnerabilities. • Can be integrated in a diverse range of tools. • Plenty of open-source tools.

6.2 Weaknesses

Table 4: Weaknesses of IAST vs. Web Fuzzing

Methodology	Weaknesses
IAST	• Can be resource-intensive. • Requires integration into the application. • Introduces operational overhead. • Limited to tested execution paths. • Expensive products, no open-source alternatives.
Web Fuzzing	• Rather high rate of false positives. • Often lacks precision in vulnerability location (in the source code). • Demands substantial computational resources. • Can be time-consuming. • Might not fully understand application logic (especially black-box fuzzing).

6.3 Synergies

The prevailing environment of web application security testing demands flexible, thorough, and accurate testing methods. Integrating both Fuzzing and IAST into the CI/CD pipeline is possible without any inherent clashes, and it is set to offer an enhanced *comprehensive vulnerability coverage*. Fuzzing acts as the initial line of defense due to its wide testing range, which can simulate different inputs and has the capability to reveal edge-case vulnerabilities and potential runtime errors. In contrast, IAST, with a deeper insight into the application's internals, excels in identifying vulnerabilities that emerge during particular execution paths, thereby complementing the broad approach of fuzzing with notable precision.

While pinpointing the exact location of vulnerabilities detected by fuzzing in the source code can be challenging, the integration of an IAST tool can assist developers in more efficiently locating these vulnerabilities. This collaborative security approach can be characterized as *dynamic and adaptive testing*. As fuzzing consistently exposes the web application to numerous inputs, IAST meticulously monitors the application's responses in such scenarios, generating detailed reports that enable developers to enhance the application's security continuously. This robust combination not only ensures that the web application remains resilient amidst unexpected conditions but also guards it against unpredictable threats and inputs.

6.4 Future Directions

1. **Machine Learning:** The advancements in artificial intelligence, particularly machine learning, could greatly improve security methods. Using AI's ability to quickly process large datasets, we might get better at automatically finding vulnerabilities in source code. Also, a well-tuned AI model could reduce the number of false alerts.

2. **Cloud and Serverless Architecture:** More companies are moving to cloud solutions and using serverless architectures for their web applications. Serverless means the code runs in response to events, and this approach has its own set of security challenges, especially around safely moving data in the cloud. It would be useful to see tools like IAST that are specially made for these cloud setups.
3. **Open-Source Availability:** While fuzzing has many open-source projects, IAST tools don't have any. The benefits of open-source tools, such as improvements from the community, being transparent, being easy to customize, and being cost-effective, highlight their value in overall security.

7 Conclusion

The domain of web applications is both intricate and continuously growing. The vast number of web applications requiring security testing surpasses what can be handled manually, prompting current research to enhance automatic vulnerability detection techniques. It's crucial for a web application to be tested not only at the end of its development but also throughout its production phase. Introducing security testing during production is cost-effective and yields superior outcomes compared to when security is merely an afterthought. In this paper, we explored two methodologies for detecting web vulnerabilities: interactive application security testing (IAST) and web fuzzing.

Our examination of IAST underscores its prowess in real-time vulnerability detection and its accuracy in vulnerability identification. Merging the strengths of SAST and DAST, IAST provides a holistic view, bridging the divide between static code analysis and dynamic runtime analysis. Its capability to swiftly detect vulnerabilities and precisely locate them within the source code positions it as a foundational methodology for crafting top-tier security tools, essential for web application development. Regrettably, there are no open-source IAST tools currently, and the commercial options are priced such that they are mainly accessible to larger corporations.

Conversely, fuzzing, with its automated, wide-ranging testing, excels in revealing edge-case vulnerabilities that might elude other techniques. Its operation, requiring minimal (or no) insight into the application's internals, ensures an impartial testing scenario. While there's a wealth of open-source fuzzers, few are tailored specifically for web fuzzing, yet the entry threshold remains relatively low. Though our paper emphasizes fuzzing as a technique rather than a tool, it's worth noting that a standalone web fuzzer, while potent, isn't wholly sufficient for tool creation. Fuzzers commonly serve in the attack phase of web application vulnerability scanners; preceding fuzzing, a web crawler identifies entry points, and post-fuzzing, a component reports the findings.

Comparing these techniques sheds light on their distinct strengths, shortcomings, and collaborative potential, especially concerning the identification of web vulnerabilities as categorized in the OWASP Top 10. While this paper provides an overview of these methods in the web application context, a key takeaway from our research is the promising potential of integrating web fuzzing with IAST for vulnerability detection. The IAST technique excels in monitoring web application internals during runtime; employing a fuzzer allows for a diverse range of inputs that might otherwise be overlooked. By dispatching strategically formulated requests, we can uncover specific scenarios less likely to be detected otherwise. Given the web fuzzer's capability to encompass a vast input spectrum, the IAST tool can observe these requests within the application, identifying not just the problematic requests but also pinpointing the associated issues in the source code.

Looking ahead, the incorporation of AI-driven algorithms, the ascent of cloud and serverless architectures, and the growing emphasis on open-source tools present fresh prospects for application security. It's evident that both IAST and web fuzzing methodologies will adapt and respond to emergent security challenges, capitalizing on technological advancements.

8 Limitations and Future Work

The research primarily concentrates on IAST and fuzzing, specifically in relation to vulnerabilities enumerated in the OWASP Top 10. As a result, other application security testing methodologies and less prominent web vulnerabilities, which might gain significance in the future, are not emphasized. Although we examined multiple tools associated with both IAST and fuzzing, it was inevitable that some tools would be excluded from our analysis. In terms of IAST, we couldn't identify any open-source tools. Consequently, most of the information regarding these tools was sourced from the software providers, raising concerns about potential bias. When evaluating web fuzzing, gauging its efficiency in detecting vulnerabilities becomes challenging. This is because fuzzing is often embedded within a broader tool, making it tricky to isolate and assess the performance of the fuzzing component alone.

For future endeavors, there is an opportunity to explore other application security testing approaches, aiming to provide a holistic understanding of the domain. The creation of an open-source IAST tool, especially for research, is a pressing requirement. Subsequent research might consider evaluating a broader range of tools based on the criteria outlined in this paper. A valuable enhancement would involve rigorously testing the efficacy of web fuzzing components across various tools. This can be achieved by supplying fuzzers with a predefined list of entry points, thereby excluding the impact of web crawlers, and juxtaposing the outcomes with established benchmarks. Lastly, it would be intriguing to validate our hypothesis that a synergistic approach, integrating fuzzing with interactive security testing, could lead to heightened security measures.

Bibliography

[1] M. Jazayeri, "Some trends in web application development," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 199–213.

[2] A. Sarkar, "Overview of web development life cycle in software engineering," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 3, no. 6, pp. 2456–3307, 2018.

[3] N. James, "AWS Penetration Testing report: Everything you should know!" 5 2023. [Online]. Available: https://www.getastra.com/blog/security-audit/cyber-security-statistics/

[4] "OWASP Top Ten — OWASP Foundation." [Online]. Available: https://owasp.org/www-project-top-ten/

[5] B. Zhang, J. Li, J. Ren, and G. Huang, "Efficiency and effectiveness of web application vulnerability detection approaches: A review," *ACM Comput. Surv.*, vol. 54, no. 9, oct 2021. [Online]. Available: https://doi-org.proxy-ub.rug.nl/10.1145/3474553

[6] M. Aljabri, M. Aldossary, N. Al-Homeed, B. Alhetelah, M. Althubiany, O. Alotaibi, and S. Alsaqer, "Testing and exploiting tools to improve owasp top ten security vulnerabilities detection," in *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2022, pp. 797–803.

[7] O. B. Fredj, O. Cheikhrouhou, M. Krichen, H. Hamam, and A. Derhab, "An owasp top ten driven survey on web application protection methods," in *Risks and Security of Internet and Systems*, J. Garcia-Alfaro, J. Leneutre, N. Cuppens, and R. Yaich, Eds. Cham: Springer International Publishing, 2021, pp. 235–252.

[8] M. Ehmer and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, 06 2012.

[9] L. Dencheva, "Comparative analysis of static application security testing (sast) and dynamic application security testing (dast) by using open-source web application penetration testing tools," Ph.D. dissertation, Dublin, National College of Ireland, 2022.

[10] B. Mburano and W. Si, "Evaluation of web vulnerability scanners based on owasp benchmark," in *2018 26th International Conference on Systems Engineering (ICSEng)*, 2018, pp. 1–6.

[11] "Seeker IAST Tool and Services — Synopsys." [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/interactive-application-security-testing.html

[12] "Contrast Assess — IAST Security Testing — Contrast Security." [Online]. Available: https://www.contrastsecurity.com/contrast-assess

[13] R. van Beckhoven, "Finding significant vulnerabilities in complex web applications using fuzzing," Jun 2022.

[14] Xmendez, "GitHub - xmendez/wfuzz: Web application fuzzer." [Online]. Available: https://github.com/xmendez/wfuzz

[15] "sqlmap: automatic SQL injection and database takeover tool." [Online]. Available: https://sqlmap.org/

[16] Y. Pan, "Interactive application security testing," in *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 2019, pp. 558–561.

[17] J. Im, J. Yoon, and M. Jin, "Interaction platform for improving detection capability of dynamic application security testing." in *SECRYPT*, 2017, pp. 474–479.

[18] H. Setiawan, L. E. Erlangga, and I. Baskoro, "Vulnerability analysis using the interactive application security testing (iast) approach for government x website applications," in *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*, 2020, pp. 471–475.

[19] A. Seth, *Comparing Effectiveness and Efficiency of Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) Tools*. North Carolina State University, 2022.

[20] F. Mateo Tudela, J.-R. Bermejo Higuera, J. Bermejo Higuera, J.-A. Sicilia Montalvo, and M. I. Argyros, "On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications," *Applied Sciences*, vol. 10, no. 24, p. 9119, Dec 2020. [Online]. Available: http://dx.doi.org/10.3390/app10249119

[21] "Jenkins." [Online]. Available: https://www.jenkins.io/

[22] "The ZAP homepage." [Online]. Available: https://www.zaproxy.org/

[23] Sonar, "Code Quality Tool and Secure Analysis with SonarQube." [Online]. Available: https://www.sonarsource.com/products/sonarqube/

[24] "OWASP Benchmark — OWASP Foundation." [Online]. Available: https://owasp.org/www-project-benchmark/

[25] Sparrow, "[Product] Sparrow SAST - Sparrow," 7 2023. [Online]. Available: https://sparrowfasoo.com/en/product/sast/

[26] A. M.-A. S. Solver, "Z3str4." [Online]. Available: https://z3str4.github.io/

[27] "The DevSecOps platform." [Online]. Available: https://about.gitlab.com/

[28] "Visual Studio Code - Code editing. Redefined," 11 2021. [Online]. Available: https://code.visualstudio.com/

[29] C. Guindon, "Eclipse Desktop Amp; Web IDEs — The Eclipse Foundation." [Online]. Available: https://www.eclipse.org/ide/

[30] "Cloud computing services - Amazon Web Services (AWS)." [Online]. Available: https://aws.amazon.com/

[31] "Cloud Computing Services — Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us

[32] W. contributors, "Google Cloud Platform," *Wikipedia*, 9 2023. [Online]. Available: https://en.wikipedia.org/wiki/Google_Cloud_Platform

[33] "Compare Seeker vs Synopsys API Security Testing." [Online]. Available: https://www.peerspot.com/products/comparisons/seeker-35912_vs_synopsys-api-security-testing

[34] "Synopsys Software Integrity Customer community." [Online]. Available: https://community.synopsys.com/s/seeker-status

[35] "Compare Seeker vs Synopsys API Security Testing." [Online]. Available: https://www.peerspot.com/products/comparisons/seeker-35912_vs_synopsys-api-security-testing

[36] "IntelliJ IDEA – the leading Java and Kotlin IDE," 6 2021. [Online]. Available: https://www.jetbrains.com/idea/

[37] "AWS Marketplace: Contrast Security- The Secure Code Platform." [Online]. Available: https://aws.amazon.com/marketplace/pp/prodview-g5df2jw32felw

[38] "Contrast supported technologies." [Online]. Available: https://www.contrastsecurity.com/security-agent

[39] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi-org.proxy-ub.rug.nl/10.1145/96267.96279

[40] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[41] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[42] I. Andrianto, M. M. I. Liem, and Y. D. W. Asnar, "Web application fuzz testing," in *2017 International Conference on Data and Software Engineering (ICoDSE)*, 2017, pp. 1–6.

[43] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box fuzzing for web applications," in *Computer Security – ESORICS 2021*, E. Bertino, H. Shulman, and M. Waidner, Eds. Cham: Springer International Publishing, 2021, pp. 152–172.

[44] X. Zhou and B. Wu, "Web application vulnerability fuzzing based on improved genetic algorithm," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1, 2020, pp. 977–981.

[45] D. Zhao, "Fuzzing technique in web applications and beyond," *Journal of Physics: Conference Series*, vol. 1678, no. 1, p. 012109, nov 2020. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1678/1/012109

[46] A. Alsaedi, A. Alhuzali, and O. Bamasag, "Effective and scalable black-box fuzzing approach for modern web applications," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 10 068–10 078, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157822003573

[47] S. Kumi, C. Lim, S.-G. Lee, Y. O. Oktian, and E. N. Witanto, "Automatic detection of security misconfigurations in web applications," in *Proceedings of International Conference on Smart Computing and Cyber Security*, P. K. Pattnaik, M. Sain, A. A. Al-Absi, and P. Kumar, Eds. Singapore: Springer Singapore, 2021, pp. 91–99.

[48] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma, J. Li, and T. Wei, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," 2023.

[49] "Testing for server-side request forgery." [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/19-Testing_for_Server-Side_Request_Forgery

[50] "Wfuzz: The Web fuzzer — Wfuzz 2.1.4 documentation." [Online]. Available: https://wfuzz.readthedocs.io/en/latest/

[51] Cuncis, "Fuzzing Made Easy: How to Use wfuzz for Efficient Web Application Testing?" 4 2023. [Online]. Available: https://medium.com/@cuncis/fuzzing-made-easy-how-to-use-wfuzz-for-efficient-web-application-testing-d843e5b089bf

[52] "WFUZZ-WEB FUZZER — briskinfosec." [Online]. Available: https://www.briskinfosec.com/toolooftheday/toolofthedaydetail/WFUZZ-WEB-FUZZER

[53] Moulik, "WFuzz Full Tutorial — Updated 2023," *TECHYRICK*, 5 2023. [Online]. Available: https://techyrick.com/wfuzz-full-tutorial/

[54] "Edge-security group - Wfuzz." [Online]. Available: http://www.edge-security.com/wfuzz.php

[55] "wfuzz — Kali Linux Tools." [Online]. Available: https://www.kali.org/tools/wfuzz/

[56] Sqlmapproject, "GitHub - sqlmapproject/sqlmap: Automatic SQL injection and database takeover tool." [Online]. Available: https://github.com/sqlmapproject/sqlmap

[57] ——, "Features." [Online]. Available: https://github.com/sqlmapproject/sqlmap/wiki/Features

[58] "sqlmap — Kali Linux Tools." [Online]. Available: https://www.kali.org/tools/sqlmap/