WWU MÜNSTER

University of Münster
School of Business & Economics

Don Benedikt Diephaus

Finding vulnerabilities by combining taint analysis and dynamic application security testing - Implementation of an IAST prototype

Bachelor Thesis

at the Research Group for IT Security
(University of Münster)

Principal Supervisor:	Prof. Dr. Thomas Hupperich
Associate Supervisor:	Tamara Sophie Gunkel, M.Sc.

Presented by:	Don Benedikt Diephaus [442808] Alsenstraße 16 48147 Münster +49 152 02767552 d_diep01@uni-muenster.de

Submission:	23rd September 2021

Contents

Figures

Tables

Listings

Abbreviations

API	Application Programming Interface
AST	Application Security Testing
DAST	Dynamic Application Security Testing
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAST	Interactive Application Security Testing
JS	Javascript
JSON	JavaScript Object Notation
OWASP	Open Web Application Security Project
PyT	Python Taint
SAST	Static Application Security Testing
SDLC	Software Development Life Cycle
SSTI	Server Side Template Injection
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface
XSS	Cross-Site-Scripting
ZAP	Zed Attack Proxy

Abstract

The aim of this bachelor thesis is the prototypical development of a novel IAST approach for Python web applications. The underlying research question is: "How can methods of SAST and DAST be combined in the form of IAST?" To answer this question, a central use case was established that illustrates the individual deployment of DAST and SAST tools onto a mock-up web application that contains various XSS vulnerabilities. The use case served to analyse the functionalities of the selected DAST and SAST tools, which were subsequently integrated into the architecture of the IAST prototype. The results of the IAST approach compared to those of the central use case demonstrate the potential of the derived concept; the search for vulnerabilities has improved in terms of quality and quantity.

1 Introduction

The Internet provided the means to create and deploy innovative web applications to millions of people worldwide. Thus, over the course of time, the relevance of web applications has steadily increased to the point of becoming an indispensable part of many people's lives. This entails an exchange of countless, often sensitive data, which poses a demand for security against malicious activities.

"While software engineering is about ensuring that certain things happen, security is about ensuring they don't" [And10, p. 4]. Precisely, IT security revolves around the protection of an asset's confidentiality, integrity, and availability from security threats within a computer system [Hup19, p. 10-15]. In the context of web applications, Cross-Site-Scripting (XSS) attacks violate the principles of confidentiality and integrity [Wal20]. Considering the constantly evolving functionalities of web applications, the identification of XSS vulnerabilities and the associated provision of IT security grows increasingly complex [Kvo20]. One way of providing IT security is through the deployment of automated Application Security Testing (AST) tools. A category of AST that is considered particularly successful in the search for security flaws is Interactive Application Security Testing (IAST), which combines the advantages of Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST) [Pan19, p. 558].

This bachelor thesis is dedicated to the research question "How can methods of SAST and DAST be combined in the form of IAST?". For this purpose, a comprehensive understanding of SAST and DAST is to be conveyed by means of their exemplary execution on a web application with XSS vulnerabilities. Subsequently, a concept for the development of an IAST prototype for Python web applications is to be determined and implemented.

The structure of this thesis is as follows: After this introduction, Chapter two follows by explaining the theoretical foundations of XSS and AST. Chapter three outlines the extraction of a concept for the IAST prototype through a central use case. In Chapter four, the implementation of the prototype is documented. Subsequently, in Chapter five, the evaluation of the prototype is conducted. Finally, Chapter six summarises the findings of this thesis and provides an outlook.

2 Application Security Testing in the Context of Cross-Site-Scripting

Targeted security attacks on web applications, such as XSS are highly prevalent [van+17, p. 13]. In order to find security issues within web applications and mitigate resulting damages, application security practices are introduced to the Software Development Life Cycle (SDLC) [MHM19]. As one approach, AST tools can be deployed throughout development, enabling the automatic identification of security-critical bugs or other vulnerabilities [Mat+20, p. 2]. The following subsections provide the necessary theoretical foundations to understand the approach of this thesis. As the implementation of the IAST prototype exclusively covers the detection of XSS vulnerabilities, initially XSS, its functionality and categories are illustrated. Afterwards, AST and its various subcategories are introduced and correlated to XSS.

2.1 Cross-Site-Scripting

According to the Open Web Application Security Project (OWASP) Top Ten List of Web Application Security Risks of 2017, XSS ranked seventh, thus qualifying amongst the most prevalent issues found in applications [van+17, p. 13]. XSS attacks imply the injection of harmful scripts into otherwise benign and trusted websites[Kir21]. An XSS vulnerability can occur wherever user input is required, e.g., in search engines that echo a user's search keywords or web message boards that allow users to post messages [Spe05, p. 2]. They arise when untrusted data is allowed to enter a web application without prior input validation [CWE06]. Alternatively, during page generation, when the application does not prevent the loaded data from containing executable content like Javascript (JS) and Hypertext Markup Language (HTML) tags, an XSS vulnerability occurs [CWE06]. By tampering with user input sources, an attacker can then place JS scripts into the server's response, enforcing the legitimacy of the script. This injection effectively circumvents the web browser's same-origin policy, granting access to any sensitive information retained by the browser [Hup20, p. 10]. From this point, the script executes on the client's browser with sometimes devastating consequences for the victim [Hup20, p. 16].

The most common XSS practices are categorized into Persistent and Non-Persistent XSS [Kir21]. Persistent, also known as Stored XSS attacks, execute malicious scripts onto the victim's system that originate from the vulnerable application's database [Kir21]. The following code snippet illustrates an example JS script for transmitting cookie data used for an XSS attack. The script generates a Hypertext Transfer Pro-

tocol (HTTP) request to a server controlled by an attacker, containing the browser's cookie within the `data` parameter.

```
1 <script>
2     new Image().src="http://attacker.com/do?data="+document.cookie;
3 </script>
```

Listing 1 Example XSS payload: Cookie data transmission script

Given an insecure message board that allows users to post messages, an attacker could publish the said script and inject its malicious payload into the underlying database. Whenever an unknowing user accesses the website, thus loads the website's assets and generates its view, the malicious code executes. As the script originates from the server's response, a trusted source, it is granted access to any sensitive information retained by the victim's browser, such as cookies and session information. Consequently, the attacker obtains the victim's cookie data.

The Non-Persistent approach also referred to as Reflected XSS, means the reflection of harmful code onto the victim's system [Kir21]. For this, the victim itself is required to transmit an HTTP request containing the script that is then reflected and executed [Kir21]. Typically, unknowing users are tricked into calling a manipulated URL via Social Engineering, or Phishing Mails [Kir21]. This URL directs to the vulnerable web application while including the harmful script as a request parameter. Once the URL is clicked, the vulnerable application is loaded and it includes the script in its view. Hence, the code from the script is executed within the trusted context between the vulnerable web application and the browser, similar to the Persistent approach.

There is a large variety of XSS exploits. Most of them involve the transmission of private data [Kir21]. Attacks like the example above aim at stealing cookie data which can include session information. Successfully executing them could enable an attacker to take over the victim's session, also known as session hijacking [Hup20, p. 20]. Furthermore, attackers could send malicious requests to a website on behalf of the victim or compromise the victim's browser and the underlying system integrity [CWE06]. In most cases, XSS is launched without the victim being aware of it [CWE06]. Preventing XSS requires the separation of untrusted data from active browser content [van+17, p. 13]. Safe programming practices, like escaping untrusted HTTP request data, will resolve most Reflected and Stored XSS vulnerabilities [van+17, p. 13]. Additionally, validating user input both syntactically and semantically to ensure only properly formed data entering the application minimizes the risk of XSS attacks [CWE06]. While these approaches are based on

programming best practices, methods to automatically detect programming flaws and vulnerabilities can be beneficial.

2.2 Application Security Testing

The provision of Application Security is a gradual process in which the security of a system steadily increases by adding security measures [And10]. The deployment of AST tools serves as one measure to removing programming flaws and thus increasing security. Moreover, AST tools offer an efficient and low-cost alternative to manual code reviews or traditional test plans while being comparably effective at finding known vulnerabilities [Sca18]. Since a large variety of open source and proprietary AST tools exists, a classification of the different AST techniques is generally made [PT 19]. In the following, the categories SAST, DAST, and IAST as well as their functionalities, are explained. Furthermore, advantages and disadvantages are ruled out, and specific methods are explained.

2.2.1 Static Application Security Testing

SAST tools statically analyze the source, binary, or byte code to identify and report potential security flaws. Thus, they operate on the unexecuted code of an application [Cro+21, p. 2]. SAST is also referred to as white-box testing, as it has access to an application's internal structure or workings [Sca18]. SAST tools employ varying algorithmic designs for scanning the source code, e.g., Abstract Interpretation, Theorem Provers, or Taint Analysis [Mat+20, p. 4]. Since they can detect vulnerabilities at the early stages of the SDLC, SAST tools are of advantage [Roh19, p. 9]. Contrary to DAST, SAST tools analyze the entire application, covering all attack surfaces [Mat+20, p. 4]. Additionally, they report security flaws by specifying the code line, making the process of eliminating errors simple and cost-effective [PT 19]. Many drawbacks of SAST tools stem from the fact that they do not understand the tested application's execution context [Roh19, p. 9]. Vulnerabilities, such as authentication problems, access control issues, or configuration errors, are difficult to identify without executing the application [OWA21]. In addition, since SAST tools cover the source code of an entire application, a significant amount of false positives (detected vulnerabilities that are not real) are commonly reported, requiring manual audits of the results [Mat+20, p. 4].

Static taint analysis, which is based on the static analysis of source code, serves as an example of a SAST method [MD19]. Static analysis implies gaining knowledge about a program's properties without executing it, enabled through the systematic

examination of the underlying source code [Vel18, p. 13]. Therefore, static analysis tools are able to analyze possible executions of the program to indicate where issues can occur [Vel18, p. 22]. Taint analysis means the analysis of information flow [Vel18, p. 26]. When searching for web application vulnerabilities, taint analysis can either be used to analyze the flow of untrusted information into a security-sensitive operation (integrity violations), or the flow of confidential information into publicly observable parts of the application (confidentiality violations) [Sri+11, p. 1053]. Considering XSS vulnerabilities that are caused by untrusted user input, the taint analysis variant for detecting integrity violations is of relevance. In this context, untrusted or tainted data is defined as information propagating in an application that is derived from untrusted external sources, e.g., the malicious request parameter data from Listing 1. In order to deploy taint analysis tools, initially, a set of security rules has to be specified [Sri+11, p. 1055]. These security rules are composed of the following parts [Sri+11; Tri+09]:

- *Sources*, which are the origin of tainted/untrusted values
- *Sinks*, which represent security-sensitive operations where tainted data terminates and potentially causes the exploit of a security vulnerability
- *Sanitizers*, which are operations that clean tainted data and thus intercept the taint flow

Based on this, taint analysis detects a vulnerability when tracking data that flows from a defined source, then propagates through data- and control flow without passing through a defined sanitizer, and ultimately terminates in a defined sink [Sri+11; Tri+09]. Given an exemplary reflected XSS taint flow, its source would be the method that extracts the content of a query string, and its sink would be the dynamic inclusion of this data within the server's response.

2.2.2 Dynamic Application Security Testing

DAST tools, contrary to SAST, operate at an application's running state. They simulate automated attacks or other unexpected test cases on an application by communicating through its front-end. For this, DAST tools first try to detect the application's attack surface to subsequently test all externally exposed source inputs. Consequently, by observing the tested application's HTTP responses, vulnerabilities and architectural weaknesses can be identified. [Mat+20, p. 4] Since DAST tools have no prior knowledge of the system they test, it is also known as black-box testing [Sca18]. The advantage of DAST tools is that realistic attack situations during

runtime are simulated. This way, flaws in application logic can be detected [PT 19]. Also, fewer cases of false positives distinguish DAST from SAST, as vulnerabilities are discovered by successfully exploiting them [PT 19]. Furthermore, since DAST tools communicate via HTTP, they can be used regardless of the server-side language or framework used by the application [Roh19, p. 9]. However, since DAST tools can only be used if an executable version of the application exists, they are deployed relatively late in the SDLC [Roh19, p. 9]. Unlike with SAST, no underlying information is provided when vulnerabilities are found [PT 19]. Another disadvantage is that usually only a few "low hanging fruits" are detected by DAST tools, meaning a high chance of false negatives (real vulnerabilities that are not detected) arises [Roh19, p. 9].

A common DAST method is the automated penetration testing of web applications. "Penetration testing can be defined as a legal and authorized attempt to locate and successfully exploit computer systems for the purpose of making those systems more secure" [EK13, p. 1]. Thus, the process of penetration testing revolves around verifying and expressing the existence of vulnerabilities and providing specific recommendations for addressing as well as fixing the security issues at hand [EK13]. For this purpose, the following fundamental process steps are carried out [ZAP21b]:

(1) Exploring the system

(2) Attacking the system

(3) Reporting the results

An automated penetration testing tool begins with the detection of external input sources by gradually exploring the available Uniform Resource Locator (URL)s of a web application. Attacks a malicious user would otherwise execute, e.g., the XSS payload from Listing 1, are then simulated. The subsequent analysis of response searches for indicators of the successful execution of an attack. Deploying an example attack payload that includes the JS method `alert`, the resulting alert box would serve as an indicator for the successful exploit of the web application. They are thus collected and reported, including details about the deployed attack payload and other useful information. [ZAP21b]

2.2.3 Interactive Application Security Testing

The term IAST is relatively new in the field of AST [Pan19, p. 558]. It implies combining methods from both SAST and DAST to introduce a new generation of application security technology [Pan19, p. 558]. To find vulnerabilities, IAST makes

use of information from within the application at runtime [WD12, p. 6]. For this, applications under test are instrumented with agents that monitor runtime activity [WD12, p. 8]. Thus, IAST tools run directly on the server. Through this, they can perform precise security analysis based upon runtime information [Mat+20, p. 5]. As IAST tools have more access to contextual information than SAST or DAST tools, they can generate better results [WD12, p. 6]. Unlike DAST, IAST tools can follow the execution of source code, data flow, and observe code that is not exposed through external interfaces [WD12, p. 6]. Contrary to SAST, they have access to the entire runtime context, including HTTP requests, database results, and values from files [WD12, p. 6]. Hence, higher testing accuracy is provided, generating fewer false positives [Mat+20; SEB20] [SEB20]. Further, IAST tools have broader code coverage and provide more accurate output, as code lines are provided with vulnerability reports [WD12, p. 10].

3 Derivation of a Concept for the Implementation of the IAST Prototype

This bachelor thesis is aimed at implementing a prototypical approach to IAST that uses the functionalities of selected SAST and DAST tools to deliver improved results in the search for XSS vulnerabilities. The question resulting from this objective is the following: "How can methods of SAST and DAST be merged in the form of IAST?" To answer this question, the following chapter, Chapter three, is dedicated to finding a concept for the implementation of the IAST prototype. For this purpose, first, a conventional approach to AST by deploying SAST and DAST individually, is presented. Hence, in Subchapter 3.1, the building blocks used for this approach are explained, and the respective results are listed. Consequently, in Subchapter 3.2, the results are interpreted, and limitations in the stand-alone use of AST tools are objectively identified and converted into realizable project goals. In conclusion, considering the previously obtained requirements for the IAST prototype, in Subchapter 3.3, a practical concept for the IAST prototype is derived, using the building blocks from the original use case.

3.1 Individual Deployment of Application Security Testing Tools

The following section depicts the central use case of AST tools. The tools that are introduced in this Subchapter serve as building blocks of the IAST prototype architecture. Thus, this section illustrates their functionalities in detail. The IAST prototype is primarily implemented to identify XSS vulnerabilities. Therefore, a sample web application containing various XSS vulnerabilities has been developed. Through the use of a SAST and a DAST tool, its vulnerabilities are to be detected. The selection of DAST and SAST tools was limited to open-source projects to ensure that the underlying source code can be examined to see how functionality is implemented. The specifications of the machine, on which this use case and its development is based on, are listed in Table 7 of the appendix. In the upcoming subsection, the development of the vulnerable web application named FlaskXSS is explained in detail. Then, the selected DAST tool OWASP Zed Attack Proxy (ZAP), its functionalities, and finally, its analysis of FlaskXSS are examined. Consequently, the SAST tool Pysa is similarly investigated.

3.1.1 FlaskXSS

The mock-up web application, containing various XSS vulnerabilities, was implemented with the Python framework Flask, justifying the name FlaskXSS. Flask is a

web application framework designed for the development of dynamic websites, web applications, or web services. It is also referred to as a microframework, meaning a lightweight framework that only includes the absolutely necessary components [Pal10b]. Owing to this, the focus lies on extensibility, and the core of an application is kept simple [Pal10b]. Flask comes with only a few built-in dependencies, including Jinja2, which serves as a full-featured template engine, and Werkzeug, a Web Server Gateway Interface (WSGI) web application library [Ghi20, p. 21]. They cover the basic functionalities of URL routing, request-, and error-handling, templating, cookies, support for unit testing, and the provision of a development server [Her19].

Flask is chosen as the framework for FlaskXSS because of its flexibility and modifiability. Furthermore, Flask is often termed a prototyping framework [Ghi20, p. 9], as it adds negligible overhead to small applications. Moreover, unlike other comparable frameworks such as Django, only a few security mechanisms are built into Flask. The templating engine, Jinja2 rules out XSS vulnerabilities caused in templates by automatically HTML escaping the outputs it generates. Auto-escaping is included for `.html`, `.htm`, `.xml` and `.xhtml` files unless explicitly disabled [Pal10c]. Apart from this, Flask requires the developer to fix security vulnerabilities as they surface, e.g., through the use of third-party extensions [Her19]. Consequently, Flask provides ideal conditions for implementing a small mock-up web application with various vulnerabilities.

The entire application logic of FlaskXSS is distributed in the two Python files, `flaskxss.py` and `db.py`. Moreover, Jinja2's auto-escaping is disabled across all template files through the use of `{% autoescape false %}` tags. Upon running FlaskXSS, Flask's local development server is deployed. Accessing the index route of FlaskXSS, which serves as its homepage, an overview of the application's functionalities as well as links to navigate to the individual vulnerable web pages are displayed.

The `persistent` HTTP handler function included in `flaskxss.py` corresponds to `GET` and `POST` requests to the `/persistent` URL. It renders the `persistent.html` template, which contains a form where users can submit comments. The template further enlists all comments that have previously been posted. When the comment form is submitted, a `POST` request is received. This triggers the `add_comment` function within the `db.py` file, which will add the form's content to the underlying SQLite3 database. From this point, the submitted message is included in all future HTML responses through the `get_comments` function call. Figure 1 puts the handler, with the corresponding database operations, into context.

```
@app.route('/persistent', methods=['POST', 'GET'])
def persistent() -> str:
    if request.method == 'POST':
        comment = request.form['comment']
        db.add_comment(comment)
    comments = db.get_comments()
    return render_template('persistent.html', comments=comments)
```

Listing (2) FlaskXSS: Persistent HTTP handler function

```
def add_comment(comment) -> None:
    db = connect_db()
    db.cursor().execute('INSERT INTO comments (comment)'
                        'VALUES (?)', (comment,))
    db.commit()

def get_comments(search_query=None):
    db = connect_db()
    results = []
    get_all_query = 'SELECT comment FROM comments ORDER BY ID DESC'
    for (comment,) in db.cursor().execute(get_all_query).fetchall()
        :
        if search_query is None or search_query in comment:
            results.append(comment)
    return results
```

Listing (3) FlaskXSS: Database operations

Figure 1 FlaskXSS: Persistent HTTP handler function and associated database operations

The described HTTP handler contains both a vulnerability for persistent as well as reflected XSS attacks. Since no input validation mechanisms are provided in the application logic, the storage of untrusted data in the application's database is enabled, accounting for the persistent XSS vulnerability. Additionally, since Jinja auto-escaping is disabled and no further escaping of request data is implemented, this untrusted data is dynamically embedded in the returned template, causing the reflected XSS vulnerability. An example exploit for such a vulnerability is given in Listing 1.

When the `/reflected` URL is called, the `reflected` HTTP handler is executed. It responds to `GET` requests by rendering the `reflected.html` view that contains a search field. According to the query string, which a user submits through this search field, comments that contain this `string` are retrieved from the database and displayed in the view. The database is populated with comments submitted through the `persistent` view. When a search query is registered, the search query parameter is passed to the `get_comments` model function, which retrieves all matching comments. In addition, the query string that led to the displayed results is reflected

on the page. Figure 4 depicts the handler function's code. The resulting database operation `get_comments` can be taken from Figure 3.

```
1 @app.route('/reflected', methods=['GET'])
2 def reflected() -> str:
3     search_query = request.args.get('query')
4     comments = db.get_comments(search_query)
5     return render_template('reflected.html', comments=comments,
          search_query=search_query)
```

Listing 4 FlaskXSS: Reflected HTTP handler function

This website contains a reflected XSS vulnerability and is additionally linked to the persistent XSS vulnerability from the previous paragraph. Untrusted data can be injected into the database via the comment section, which is then retrieved and rendered without prior HTML escaping. The vulnerability for reflected XSS attacks is caused by the dynamic inclusion of the query string within the rendered view. Since no validation mechanisms are deployed for the data entered into the search field, and the query string is not HTML escaped, malicious code could be reflected back to the user. Thus, a victim could be tricked into clicking a manipulated URL similar to the URL depicted in Listing 5.

```
1 localhost:5000/reflected?query=<script>alert("XSS")</script>
```

Listing 5 Example XSS payload: Manipulated URL containing Javascript

The website that is accessed when the `/insufsanitize` URL is entered serves as a demonstration of the insufficient validation and sanitization of user input. The handler function expects `GET` requests that contain a URL parameter `q`. Comparable to the `reflected` view, the content of the `q` parameter is displayed within the `insufsanitize.html` view. Consequently, the URL `/sufsanitize?q=Hello+World` would result in the words "Hello World" being displayed on the website. The view does not contain any additional functionalities. The handler function, however examines the value of the parameter `q` for `strings` that are associated with JS and HTML. Once the `strings` "<script>" or "alert" are included, the view's text box displays "Attack detected". Listing 6 depicts the described function.

Like the `reflected` handler function, this website contains a vulnerability for reflected XSS that is caused by the dynamic inclusion of a URL parameter within the generated view. The implemented security measure that detects certain JS and HTML elements can be circumvented by using alternate XSS syntax. XSS attacks can be conducted without the use of `<script>` tags or the `alert` method. The example depicted in Listing 1 could be executed using `<body onload=` or `<b`

```
@app.route('/insufsanitize', methods=['GET'])
def insufsanitize() -> str:
    input = request.args.get('q')
    if not input:
        response = "No input!"
    elif "<script>" in input:
        response = "Attack detected!"
    elif "alert" in input:
        response = "Attack detected!"
    else:
        response = "Input: " + input
    return render_template('sanitize.html', response=response)
```

Listing 6 FlaskXSS: Insufficient sanitization HTTP handler function

`onmouseover=` instead. This handler function is added to FlaskXSS with the consideration that DAST tools might not detect it as a vulnerability due to their high rate of false negatives, as stated in Chapter 2.2.2.

The `sufsantize` HTTP handler function serves as the sufficiently sanitized equivalent to the `insufsanitize` function. Their view's functionalities are identical. The `sufsantize` function, however sanitizes the content of the URL parameter `q` before display, using the `html.escape` function. It encodes the characters "&", "<" and ">" into sequences that cannot be interpreted as code [Pyt21]. Additionally, an alternative function to generate a view, `render_template_string`, was used. Rather than generating output from a template file, it returns a `string` that is passed as a function argument [Mak21]. The code can be derived from Listing 7.

```
@app.route('/sufsanitize', methods=['GET'])
def sufsanitize() -> str:
    input = request.args.get('q')
    if not input:
        response = "No input!"
    else:
        response = "Input: " + html.escape(input)
    return render_template_string(response)
```

Listing 7 FlaskXSS: Sufficient sanitization HTTP handler function

As previously outlined in Chapter 2.1, escaping untrusted HTTP request data serves as an effective technique for preventing XSS attacks. Although further security measures, such as the validation of user input, should be employed, it prevents the reflected XSS vulnerability represented by the `/insufsanitize` handler function. This function was included to provide the possibility of a SAST tool indicating a false positive for XSS.

When calling the `/xssheader` URL, the handler function responds to `GET` requests by returning the HTTP Referer header of the request. This header contains the URL of the website that makes the request. Meaning, when following a link, the Referer header's value contains the address of the website displaying the link. Thus, the Referer header allows servers to identify where clients originate from [Moz21]. The `xssheader` HTTP handler function deploys the `make_response` function to generate its view, converting the Referer header's content into a response object [Pal10a]. Consequently, the view prints the Referer HTTP header's content of the `GET` request it receives. The implementation of this handler function is depicted in Listing 8.

```
1 @app.route('/xssheader', methods=['GET'])
2 def xssheader() -> str:
3     referer_header = request.headers.get('Referer')
4     if not referer_header:
5         referer_header = "No Referer!"
6     response = referer_header + "<br><a href='javascript:history.
          back()'>Go back</a>"
7     return make_response(response)
```

Listing 8 FlaskXSS: XSS header HTTP handler function

This handler function contains a vulnerability for reflected XSS that can be exploited by deploying an alternative, more elaborated XSS payload. Since common XSS techniques are often anticipated and thus prevented, malicious users look for ways of injecting code into areas that are commonly overlooked by developers [Acu14]. Instead of placing scripts into the request's URL parameters, they are transmitted via an HTTP header field. A command-line tool like `cURL`, which allows to send custom HTTP requests, can be used. Therefore, such attacks depend on the improper return of header values within the server's response. The `xssheader` handler function serves to illustrate such a vulnerability.

3.1.2 OWASP ZAP

The DAST tool selected for the central use case is OWASP ZAP. ZAP is an open-source automated web application penetration testing tool [ZAP21b]. It is written in Java and available across all major operating systems [ZAP21a; MK15]. OWASP ZAP was chosen for this setting because it is an open-source project. Moreover, OWASP provides extensive documentation for ZAP, especially with regard to its API. Furthermore, in several studies, ZAP is favoured over other comparable open-source or free DAST solutions, based on factors such as usability, runtime, stability, and the accuracy of results [HN17; Sag+18]. The central component of OWASP ZAP

is a proxy that is able to intercept and inspect communication between a client and a server. When needed, ZAP modifies the content of a message and forwards it to its destination [ZAP21b]. ZAP has a large variety of features. In the following description of its functionalities, the focus is set on features that are relevant for this use case.

Performing an automated scan begins with the *exploration of the system*. For this, ZAP will deploy one of its "spiders", a tool used to automatically discover new URLs on a site that was specified by the user. The traditional spider discovers links by making requests to a URL, parsing the HTML of the response, thus enlisting the hyperlinks found within it. This process is called web crawling and continues recursively until no new resources are found. [ZAP21i] Every website ZAP finds while crawling the web application is passively scanned. Passive scanning means that HTTP messages proxied through ZAP are examined for security-relevant factors, e.g., for the absence of X-Frame Options Header or the usage of known vulnerable JS libraries [ZAP21g]. It implies that no changes are applied to any HTTP request or response [ZAP21h]. By passive scanning, ZAP will find some vulnerabilities, establish a sense for the security state of the web application and locate resources where more investigation is warranted [ZAP21b].

The next step, *attacking the system*, calls for active scanning. ZAP will attempt to find and exploit vulnerabilities by using known attack vectors against the specified target [ZAP21b]. As this puts the target at risk, it is only allowed to actively scan applications for which explicit permission has been given [ZAP21d]. ZAP proceeds to apply the active scan rules, which contain the definitions of various attack vectors. The active scan rule for reflected XSS analyzes websites by first submitting safe values wherever input is allowed. It then proceeds by analyzing the locations in which the safe value occurs within the response. Through this, ZAP determines the origin and destination of user-controlled data. Afterwards, it performs a series of attack payloads targeted at the destination. The rule for persistent XSS starts by submitting a safe but unique value. Then, the application is searched for locations where this unique value occurs. Then a series of attacks is performed, and the target locations are investigated. During active scanning, several other rules are applied, such as rules for SQL Injection or Buffer Overflow. [ZAP21e]

In the final step, *reporting the results*, ZAP links the deployment of the active scan rules with the manipulation of the web application's behavior and generates a collection of alerts [ZAP21b]. An alert is a potential vulnerability recognized by ZAP, which is associated with a specific request. They are also raised according to the results of passive scanning. Alerts contain insightful information about vulnerabilities

at hand, such as the exact name of the vulnerability, the URL that was attacked, the attack payload that led to its exploit, the affected parameters, a generic description as well as methods of mitigation [ZAP21f]. Furthermore, they are categorized hierarchically based on the risk level of the vulnerability ranging from three, being the highest detectable risk, to zero.

In order to run OWASP ZAP on FlaskXSS, the corresponding Installer file was downloaded, and the application was installed. It should be noted that repeated active scanning leads to large amounts of malicious data stored in the FlaskXSS database. As this will influence ZAP's results due to changes to the Document Object Model (DOM), the FlaskXSS `database.db` file is deleted after each execution of ZAP. Furthermore, a new ZAP session is launched for every run. Upon starting ZAP, the "Quick Start" tab within the "workspace" window is opened, revealing two scanning options. From there, the "Automated Scan" button is clicked, whereupon the URL of the target application and the web crawling type must be specified. The FlaskXSS root URL is entered, a check-mark is set for "Use traditional spider" and the "Attack" button is clicked. ZAP's spider then proceeds to crawl FlaskXSS and enlists all found URLs within the "Spider" tab inside the "information" window. The results can be derived from Table 1.

URI	**Method**
http://localhost:5000/robots.txt	GET
http://localhost:5000/sitemap.xml	GET
http://localhost:5000/	GET
http://localhost:5000/persistent	GET
http://localhost:5000/xssheader	GET
http://localhost:5000/reflected	GET
http://localhost:5000/insufsanitize (q)	GET
http://localhost:5000/sufsanitize (q)	GET
http://localhost:5000/static/styles.css	GET
http://localhost:5000/reflected (query)	GET
http://localhost:5000/persistent ()(comment)	POST

Table 1 ZAP: Crawling results

After the crawling process is finished, the "Active Scan" tab opens, and ZAP proceeds to sequentially log and execute all HTTP requests required for the active scan rules. After the analysis of FlaskXSS is finished, the results are displayed in the "Alerts" tab. The relevant persistent and reflected XSS alerts are listed in Table 2. Other vulnerabilities, e.g., caused by the absence of Anti-CSRF Tokens, X-Frame-Options Headers, or identified DOM-based XSS alerts are identified by ZAP but out of scope for this thesis. The file `ZAP_complete_report.json` within the

`assets_thesis` directory of the IAST prototype contains all vulnerabilities identified by ZAP in the context of this use case.

Category	URL	Method	Parameter	Attack
XSS (Persistent)	http://localhost:5000/persistent	POST	comment	<script>alert(1);</script>
XSS (Reflected)	http://localhost:5000/reflected?query=%3Cimg+src%3Dx+onerror%3Dalert%281%29%3B%3E	GET	query	<img src=x onerror=alert(1);>
XSS (Reflected)	http://localhost:5000/persistent	POST	comment	<script>alert(1);</script>

Table 2 ZAP: Active scanning results

3.1.3 Pysa

As the SAST tool, the Python taint analysis tool Pysa was selected. It is included as a module within the open-source Python type-checking tool Pyre. Pysa is an abbreviation for Python static analysis. It employs static taint analysis to identify potential security issues [Fac21a]. As part of the Pyre project, developed by the Facebook Open-Source community, Pysa was first open-sourced in 2018 [Lin21]. Through its use at Facebook, 44% of Instagram server issues were detected in the first half of 2020 [Lin21]. The reason for Pysa being chosen as SAST component is, being that it is currently the only maintained open-source static taint analysis tool for Python applications. The only taint analysis equivalent, Python Taint (PyT), is no longer maintained as of March 2020, and the developers recommend the use of Pysa [PyT20].

The functionalities and components of Pysa that are relevant for the central use case are explained in the following, using the analysis of a simple code example. Listing 9 contains a simplified snippet from the implementation of the FlaskXSS `/xssheader` handler function; thus, according to Chapter 2.2.1 it depicts an XSS taint flow.

```
1 @app.route('/xssheader', methods=['GET'])
2 def xssheader() -> str:
3     referer_header = request.headers.get('Referer')
4     return make_response(referer_header)
```

Listing 9 Pysa: XSS taint flow handler function

The `request.headers` class attribute is a user-controlled source of data that is regarded as a taint source. The tainted data propagates throughout the `xssheader` handler as it is passed to the `referer_header` variable. The taint flow terminates within the `make_response` function, which generates a view containing the tainted value. As stated in Chapter 3.1.1, this is considered an XSS vulnerability.

In order to identify this taint flow, Pysa operates on the specification of *security rules*. Security rules are composed of one `taint.config` file and multiple `.pysa` files. The `taint.config` file is a JSON document where sources, sinks, and rules are defined. The `.pysa` files serve as *models* that annotate the code to be analyzed with the sources and sinks defined in the `taint.config` file. Given the vulnerable code presented in Listing 9, the `taint.config` file must be defined according to Listing 10 for Pysa to recognize the taint flow. [Fac21a]

```
{
"sources": [{
        "name": "UserControlled",
        "comment": "used to annotate all data that is controllable
           by the user making a request"
    }],
"sinks": [{
        "name": "XSS",
        "comment": "used to annotate where data can cause XSS"
    }],
"rules": [{
        "name": "XSS",
        "code": 5008,
        "sources": ["UserControlled"],
        "sinks": ["XSS"],
        "message_format": "Data from [{$sources}] source(s) may
           reach [{$sinks}] sink(s)"
    }]
}
```

Listing 10 Pysa: taint.config definitions

The correlation of the `taint.config` taint source and the `request.headers` class attribute is represented by the `.pysa` model, shown in Listing 11. The declaration of `.pysa` source models is composed of the "fully qualified name" of a function or an attribute, followed by the `TaintSource[SOURCE_NAME]` decorator [Fac21a]. `headers` is a `BaseRequest` class attribute, defined in the Flask module `werkzeug.wrappers`, which explains the constitution of the fully qualified name. Subsequent decorators are set according to the syntax for type annotations in Python 3 [Fac21a]. Using the depicted model declaration in combination with the `taint.config` taint source definition, the `headers` class attribute is recognized as a source of user-controllable data; thereby it is dangerous.

```
1 werkzeug.wrappers.BaseRequest.headers: TaintSource[UserControlled]
```

Listing 11 Pysa: .pysa source declaration

The sink definition of the `taint.config` file is correlated to the `make_response` function through the `.pysa` model definition depicted in Listing 12. Comparable to the source model declaration above, sink models are composed of the fully qualified name of a function or class attribute and the setting of a `TaintSink[SINK_NAME]` decorator [Fac21a]. In this case, potentially harmful data is passed to the `*args` argument of the `make_response` function. Thus, following the Python 3 type annotations syntax, the `*args` argument is annotated with the `TaintSink[SINK_NAME]` decorator. Thereby, the `*args` argument of the `make_response` function is declared as a potential sink of user-controllable data, which may lead to XSS.

```
1 def flask.helpers.make_response(*args: TaintSink[XSS]): ...
```

Listing 12 Pysa: .pysa sink declaration

Lastly, an XSS rule has been declared within the `taint.config` file. Rules specify which sources and sinks constitute a certain taint flow and serve to output the found results within the console after the execution of Pysa. They are composed of a name, an ID, a set of sources, and a set of sinks. Furthermore, a short message explaining the found vulnerability is added. [Fac21a]

Additionally, sanitizers that intercept a taint flow can be defined to be recognized by Pysa within the `taint.config` file. Considering the `html.escape` function, utilized in Listing 7, declaring this sanitizer would prevent the potential output of a false positive. Similar to sources and sinks, sanitizer functions are correlated to the `taint.config` declaration by a `.pysa` model. The example model declaration of the `html.escape` function as a sanitizer is shown in Listing 13. [Fac21a]

```
1 @Sanitize
2 def html.escape(s, quote): ...
```

Listing 13 Pysa: .pysa sanitizer declaration

In order to run Pysa, it must first be initialized by running the `pyre init` command inside the project directory where the taint analysis is to be performed. Pyre will proceed to create a global `.pyre_configuration` file inside the root directory. [Fac21c] When starting Pysa, only the code within the repositories specified in the `.pyre_configuration` file is analyzed. Code within dependencies is not considered. Pysa assumes that tainted data flowing through library functions or any functions

Pysa has no access to is still tainted after return. This generally results in an increased number of false positives. Moreover, Pysa only considers the `taint.config` and `.pysa` files from repositories that are defined in the global configuration file. To achieve informative results, it is recommended to provide user-supplied sources, sinks, and sanitizers prior to running Pysa. Nevertheless, predefined security rules for common Python libraries are included within the Pyre-Check project directory, which allows the out-of-the-box execution. [Fac21a]

Since Pysa exclusively runs on Unix-based operating systems [Fac21b], it was installed on the Ubuntu Windows Subsystem for Linux. A complete documentation of the Pysa installation steps is contained in the `README.md` file within the IAST prototype project directory. No user-defined security rules are provided as part of the central use case. For the out-of-the-box deployment of Pysa, `taint.config` and `.pysa` model files are sourced from the Pyre-Check GitHub repository. An instruction for reproducing the taint analysis results obtained in the use case is also included in the `README.md` file. Ultimately, the command `pyre analyze --no-verify` is executed within the FlaskXSS root directory to start the taint analysis process. The results can be taken from Table 3.

Name	Code	Location	Line	Description
XSS	5008	flaskxss.sufsanitize	170	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
Potential Server Side Template Injection	6073	flaskxss.sufsanitize	170	"Potential Server Side Template Injection [6073]: User-controlled data may eventually flow into a Server Side Template Injection sink"
XSS	5008	flaskxss.xssheader	211	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"

Table 3 Pysa: Taint analysis results

3.2 Requirements Analysis in the Context of the Evaluation of Results

The use case from Chapter 3.1 represents the security testing of FlaskXSS by individually deploying the selected DAST and SAST tools with as little manual configuration as possible. ZAP was run by invoking the Automated Scan from the "Quickstart" panel, and Pysa was executed using the predefined rules and models from the Pyre-Check project directory. For this reason, deficits in the quality of the AST results can be observed. The aim of this chapter is to present the technical reasons for these shortcomings and thereby derive requirements for the implementation of the IAST prototype. Initially, an interpretation of both ZAP's and Pysa's

results is made. Afterwards, limitations are derived and converted into realizable project goals.

By crawling FlaskXSS with ZAP's traditional spider, all websites and their URL parameters were correctly identified. Based on this, ZAP detected the XSS vulnerabilities of both `/reflected` and `/persistent` views. Furthermore, ZAP's results do not contain any false positives; an expected output since DAST tools detect vulnerabilities by successfully exploiting them, granting a low rate of false positives.

Also, in line with the findings in Chapter 2.2.2, a high rate of false negatives is noted. Illustrated by the requests proxied to the `/insufsanitize` URL, the utilized attack vectors either include `alert` tags, `<script>` tags or both within the transmitted URL parameters. A list of all requests ZAP proxied to FlaskXSS is found in the `assets_thesis` folder of the IAST prototype project directory by the name `ZAP_ascan_requests_reference.csv`. The attack payload deployed by ZAP is intercepted and sanitized by the "validation mechanism" of the handler function. Therefore, the high rate of false negatives indicates that the XSS payload of ZAP's active scan rules is not diverse enough to bypass certain safety precautions. Due to this, the vulnerability from the `xssheader` handler function was not found as an uncommon method of XSS must be deployed in order to exploit it. ZAP did not transmit any XSS payload within the Referer Header, hence leading to this false negative. In fact, ZAP only sent two requests to the `/xssheader` HTTP handler, as no other possibilities of external input were identified.

The limitations identified through the use of OWASP ZAP are derived from the fact that the XSS vulnerabilities of only two out of four handler functions have been detected. Thus, the results of the use case reinforce the statements from Chapter 2.2.2, claiming a low rate of false positives as well as a high rate of false negatives.

Through the deployment of Pysa, without the provision of user-defined security rules, three vulnerabilities in the FlaskXSS codebase were found. Pysa was able to positively detect the XSS vulnerability of the `xssheader` handler function. Furthermore, the Server Side Template Injection (SSTI) vulnerability of the `sufsanitize` handler function is a true positive. However, this result is out of scope for this thesis. Lastly, Pysa issued a false positive by indicating an XSS taint flow in the `sufsanitize` method. Even though the amount of false positive results is higher compared to ZAP, Pysa's results stand in contrast to the statements in Chapter 2.2.1, since neither a particularly high rate of false positives nor a high rate of results, in general, is recognized. Pysa has been instructed to use the `taint.config` and `.pysa` files found within the `stubs/taint` subdirectory of a cloned version

of the Pyre-Check project repository. Source and sink models defined specifically for Flask modules, being the ones primarily used for FlaskXSS, are found within the `flask_sources_sinks.pysa` file of the `taint/core_privacy_security` subdirectory. Within the same directory, a comprehensive `taint.config` file is included. The security rules defined in the two files show that they are insufficient for the analysis of FlaskXSS. For comprehensibility, the `flask_sources_sinks.pysa` file is included in the `assets_thesis/pysa_security_rules` folder, and the `taint.config` can be found in the `app/Pysa` directory of the IAST prototype project directory. By referencing these files within the global Pysa configuration file, the results obtained in this use case can be reproduced.

The `persistent` handler function's taint source is the `BaseRequest.form` class attribute defined as a `UserControlled` source within the `flask_sources_sinks.pysa` model file. The handler's taint sink is the `render_template(**context)` function argument which is declared as a `ReturnedToUser` sink. Within Pysa's predefined `taint.config` file, no rule that detects a taint flow originating from a `UserControlled` source, terminating in a `ReturnedToUser` sink is declared. The defined rule for XSS taint flows expects a `UserControlled` source to terminate either in an `XSS` or `StringMayBeHTML` sink. Consequently, Pysa does not consider the `render_template` function to be a security-sensitive operation in the context of XSS.

Like the `persistent` handler function, the `reflected` and `insufsanitize` functions retrieve tainted data from the `BaseRequest.args` class attribute. This is defined as a `UserControlled` taint source. The taint flow terminates within the `render_template(**context)` function argument. Thus, for the same reason as for the `persistent` handler, these taint flows are not recognized. The assumption that Pysa does not consider the `render_template(**context)` function argument as an XSS sink, because the Jinja2 template engine escapes HTML output by default, is made. Accordingly, in the default case, no XSS risk results from the use of the `render_template()` function.

The `xssheader` handler function's XSS vulnerability was correctly detected since the `BaseRequest.headers` class attribute is declared as a `UserControlled` taint source. Furthermore, the `flask.helpers.make_response(*args)` function argument is defined as an XSS sink. Hence, the XSS rule from the `taint.config` file is able to correlate this taint flow.

The incorrectly detected taint flow of the `sufsanitize` method occurred due to the retrieval of tainted data through the `BaseRequest.args` class attribute, followed by

the use of the `render_template_string()` function. Within Pysa's model file, the `render_template_string(source)` function argument is declared as an XSS sink. Since the `html.escape` is not declared as a sanitizer, Pysa does not recognize the interception of this taint flow. Hence, the taint flow is falsely detected.

The XSS taint flow of one out of four XSS vulnerable handler functions was correctly identified by Pysa's taint analysis. Furthermore, one false positive taint flow was detected. This represents a significant limitation of the "out-of-the-box" usage of Pysa. The statement made in Chapter 2.2.1 that DAST tools report a high rate of false positives is partly reinforced. In contrast to the findings in Chapter 2.2.1, a critical amount of false negatives is presented.

Chapter 2.2.3 states that the combination of DAST and SAST techniques enables a more precise method of security analysis in the form of IAST. For this purpose, the information collected during the runtime of an application is to be merged with the static analysis of source code. It appears that the use of ZAP has led to expected results, while the use of Pysa led to unexpectedly insufficient results. This is due to the missing provision of intelligent user-defined security rules for Pysa. A fundamental requirement for the functionality of the IAST prototype is the improvement of AST results. This raises the functional requirement of enhancing the taint analysis results of Pysa. With regard to the implementation of this functionality, the technical requirement of dynamically generating Pysa models is set. This can be achieved by incorporating information generated by OWASP ZAP, which is then included in the process of taint analysis; corresponding to the operation of an IAST tool.

3.3 Architecture

As outlined in Chapter 2.2.3, IAST tools are able to achieve a higher number of accurate results through the incorporation of additional contextual information. Secondary literature suggests that such information is retrieved through the instrumentation of sensors or agents within an application that monitor runtime activity, including control flow, data flow, and architecture [WD12]. This thesis presents an approach to generating supplementary contextual information through the execution of OWASP ZAP while monitoring control and data flow. Thus, FlaskXSS has to be instrumented with a tracing system that will enable the reconstruction and monitoring of process steps triggered during penetration testing [Shk19b, p. 14].

For this task, the method of distributed tracing has been chosen. Contrary to traditional monitoring, "distributed tracing takes a request-centric view" and captures

details about the execution of related activities while a system processes a given request. Such tracing tools enable the monitoring of distributed context propagation [Shk19a]. This means they are able to attach contextual metadata to requests and pass it around during execution [Shk19b, p. 14]. They can be instrumented to record specific points of interest, such as the request URL or the value of relevant parameters [Shk19b, p. 14]. Through this, tracing systems are able to reconstruct the whole path of the request, generating a graph of events called a *trace* [Shk19b, p. 14].

To achieve the described functionality, a *tracing backend* has to be run, which collects traces from FlaskXSS using the OpenTracing API. It is a vendor-neutral API specification [Hög20] that allows developers to add distributed tracing instrumentation to Python microservices [Ope21b]. According to the OpenTracing specifications, a trace is the description of a transaction as it moves through a system [Ope21b]. It consists of multiple *spans*, which are the logical representations of a piece of workflow [Ope21b]. Spans are composed of an operation name, a start- and a finish time, and developers can add key-value *tags* to spans to provide additional information about certain operations [Ope21a]. To do so, OpenTracing offers libraries to insert tracing instrumentation into chosen points of an application's codebase [Shk19b]. This instrumentation generates data during execution which consequently is collected, arranged, correlated, and combined into a trace, using a tracing backend [Shk19b, p. 66]. The chosen open-source tracing backend, compatible with OpenTracing, is Jaeger. Jaeger provides a Graphical User Interface (GUI) that visualizes trace data as well as several Application Programming Interface (API)s that enable the retrieval and further analysis of trace data [The21b].

Hence, through the instrumentation of FlaskXSS with Jaeger, additional contextual information about the process steps caused during the execution of OWASP ZAP can be derived. The alerts that are obtained after the active scanning process represent hints about vulnerable code-fragments. Correlating both sets of information, hence linking attack vectors that lead to a definite exploit with their respective traces, will generate supplementary information. This information will disclose the FlaskXSS control and data flow that constitute the vulnerability.

Based on this, the following process steps result from the execution of the IAST prototype, which are illustrated in the sequence diagram in Figure 2. First, the process of automated penetration testing is started by communicating with OWASP ZAP. This will trigger all necessary process steps for the active scanning process, explained in Chapter 3.1.2. This includes the use of ZAP's traditional spider to identify all available resources of FlaskXSS as well as the proxying of the active scan attack pay-

loads. The initiated process steps within the FlaskXSS HTTP handler functions are simultaneously observed and collected by an instrumented tracing backend. After completion of the active scanning process, the alerts generated by ZAP are retrieved, and the tracing process ends. Consequently, ZAP's results are filtered for persistent and reflected XSS alerts, which are within the scope of this thesis. Then, traces are queried via a Jaeger API. Once the traces have been retrieved, the spans corresponding to ZAP's alert data must be extracted. This requires that the spans are instrumented in a way that they can be distinguished and categorized based on the information obtained from ZAP's alerts. Since Jaeger allows the tracing of data values processed throughout a handler function, a possibility of connecting alert data and its corresponding spans is by filtering spans according to the attack payload of an XSS alert. Hence, the data flow of this attack payload and its associated control flow becomes apparent when inspecting the respective span. This will reveal the operations associated with the retrieval of malicious data, the way malicious data propagates within the handler function as well as the operations that malicious data terminates in. Thereby, the taint flow of the exploit of an XSS vulnerability caused by OWASP ZAP is portrayed within the span. Based on this, a distinction between sources and sinks is drawn, and security rules for Pysa are generated. They are written in the form of source and sink models within a `.pysa` model file. Finally, the taint analysis is performed by Pysa and, with the help of the dynamically written security rules, more insightful results are obtained. The assumption is made that Pysa will then be able to identify all present vulnerabilities of FlaskXSS.

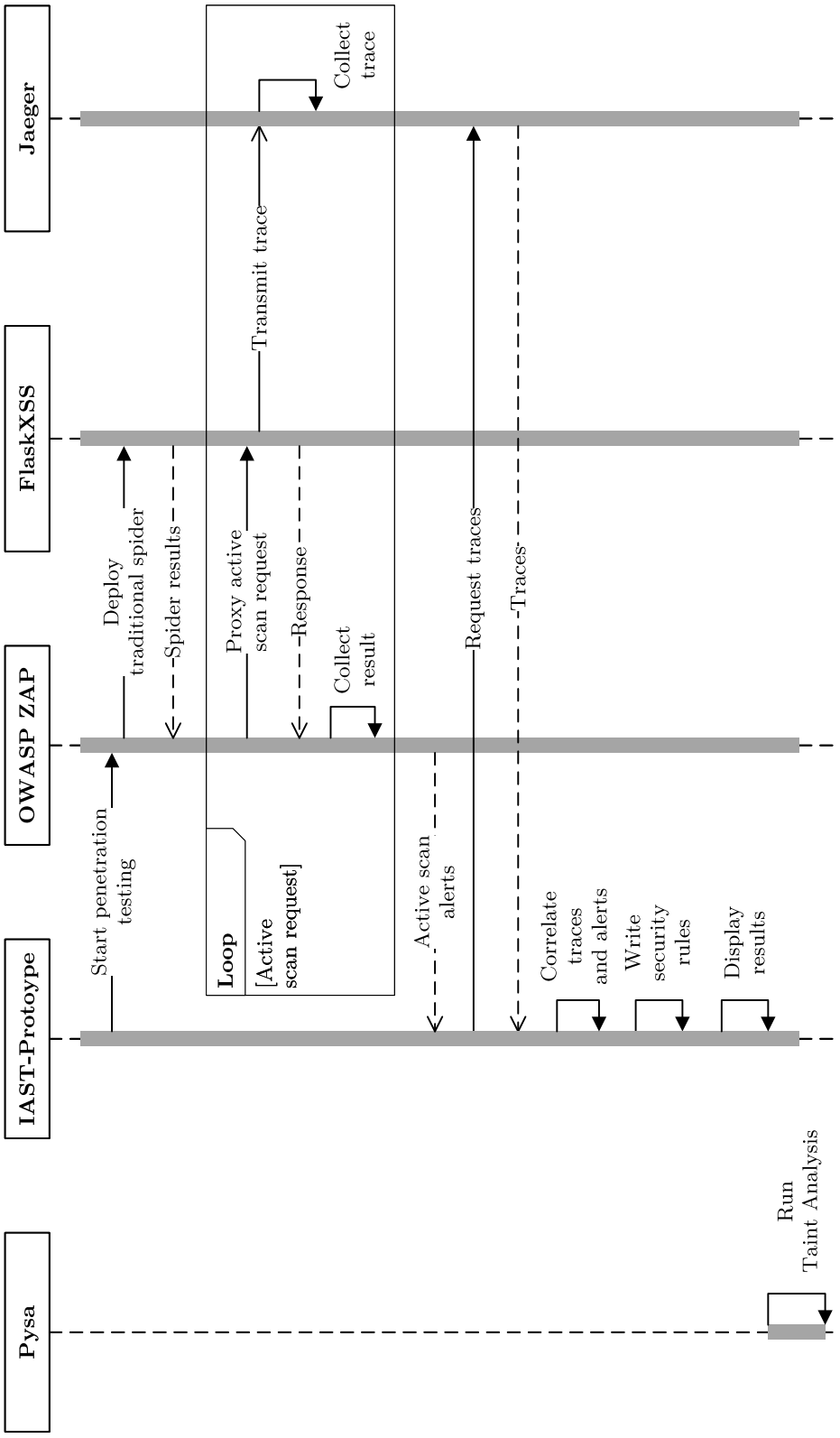

Figure 2 IAST prototype sequence diagram

4 Implementation of the IAST Prototype

The following chapter documents the implementation of the concept described in Chapter 3.3 to fulfill the requirements outlined in Chapter 3.2. Similar to the implementation of FlaskXSS, the IAST prototype is based on the Python web framework Flask. In this context, Flask is primarily used to create the GUI.

The project's source code is located in the `iast-prototype/app` package directory. Within the project root directory, the `iast_prototype.py` file serves as the starting point to run the Flask application. For the sake of completeness, the FlaskXSS project directory is included in the `iast-prototype` project directory. However, this is not mandatory for the IAST process. The base application logic within the `app` package directory is separated into HTTP handler functions, found in the `routes.py` file, configuration classes within the `config.py` file, and web form classes within the `forms.py` file. Any other Python scripts or files that are related to the operation of Jaeger, Pyre or ZAP are located in the respective subdirectories `Jaeger`, `Pyre` and `ZAP`. Additionally, the `output` directory is located within the `app` package, which contains all results collected throughout the IAST process, stored in individual JavaScript Object Notation (JSON) files. Lastly, the `static` and `templates` directories contain files necessary to generate the prototype's front-end, such as template files and styling sheets.

In the course of the programming work for the IAST prototype, three key features were identified for implementation. Firstly, runtime monitoring had to be instrumented in order to enable the tracing of ZAP attack payloads, and the retrieval of both traces and ZAP alerts. Secondly, to generate contextual information, several filtering mechanisms for both traces and ZAP alerts had to be implemented. The last key feature was to dynamically write models to a `.pysa` model file. The following subchapters illustrate the implementation of the listed features with respect to their order. Starting with the implementation of DAST runtime monitoring, the instrumentation of tracing is documented in Chapter 4.1, and the integration of ZAP's active scanning process is depicted in Chapter 4.2. Subsequently, the implementation necessary to obtain supplementary contextual information by filtering ZAP's alerts in Chapter 4.3, as well as retrieving and filtering tracing data in Chapter 4.4, are demonstrated. Thereafter, the realisation of functionalities for the dynamic creation of Pysa security rules as well as its logic is explained in Chapter 4.5. Conclusively, the execution of the IAST prototype is presented in Chapter 4.6, bringing all three key-features into context and stating the results.

4.1 Instrumentation of Tracing

In order to instrument FlaskXSS with a tracing system, a tracer first needs to be instantiated within the FlaskXSS codebase. The OpenTracing API libraries provide a mechanism for defining a global tracer within an application. The tracer instance initialization for FlaskXSS is defined in the `tracing.py` module within the FlaskXSS root directory. The `init_tracer` function expects a service name as a parameter, populates the Jaeger-specific Config class, and returns a global instance of the Jaeger tracer. The implementation of this function is derived from Chapter four of [Shk19b, p. 104]. Thus, this tracer instance is used throughout the FlaskXSS application to start new spans and export finished spans to a tracing backend. [Shk19b]

A new span is to be generated for each HTTP request handled by FlaskXSS. Therefore, all HTTP handler functions within the `flaskxss.py` file are instrumented with the `tracer.start_span()` function call. This initializes a new span that represents a set of operations within FlaskXSS. They are started within a `with` statement; thus, spans are closed automatically as soon as the HTTP handler function's logic, which is nested inside the `with` statement, terminates. Every started span is given an operation name that is set to the name of the route of the respective handler function in FlaskXSS. The naming of spans is crucial for the later correlation of ZAP alerts and spans.

In order to trace details about the control- and data-flow of each handler function, custom information about the operations a span represents must be added. Consequently, tags, which add key-value pairs of metadata to a span, are defined. Moreover, to later correlate spans with the data of ZAP alerts, the URL of each HTTP request and its request method are added as tags to every HTTP handler function's span. For this, the `trace_request_url_method` function was written within the `tracing.py` file. This function is called at the beginning of every HTTP handler function and adds tags with the keys "url" and "method", containing the values of the `request.url` and `request.method` class attribute. Hence, at the beginning of each handler function, the same set of tags containing a request's called route, its URL, and its method is provided. Listing 14 illustrates the instrumentation mechanisms mentioned using a pseudo handler function.

Moreover, all operations that request data surpasses in the course of an HTTP handler function, as well as the value of this request data, must be traced. The instrumentation of such tags was implemented manually. However, in regard of the automated tracing instrumentation, a heuristic was developed which is applied

```
@app.route('/route', methods=['POST', 'GET'])
def handler_function() -> str:
    with tracer.start_span('/route') as span:
        trace_request_url_method(request, span)
        data = operation(data)
        return render_template('template.html', data=data)
```

Listing 14 OpenTracing span initialization

equally across all handler functions. Its rules can be derived from the pseudo-code Listing 15.

```
def generate_tag(data, operation, span):
    if operation.Data_derived == True:
        Direction_decorator = 'OUT::'
    elif operation.Data_passed_to_function == True:
        Direction_decorator = 'IN::'
        Name_of_function_argument = str(get_function_argument(
           operation, data) + '::')

    fully_qualified_name = str(get_fully_qualified_name(operation))

    key = Direction_decorator + fully_qualified_name
    if Name_of_function_argument:
        value = Name_of_function_argument + str(data)
    else:
        value= str(data)

    span.set_tag(key, value)
```

Listing 15 Heuristic for the instrumentation of tags

The main objects of interest are a data object and an operation related to this data object, which are to be traced as part of a key-value pair tag. In this context, data objects include variables, function return values, or class attribute values. In order to avoid overhead, only operations that are related to dynamic data objects are traced. The mere rendering of a static template, as an example, is not traced. All tracing rules outlined in the following are also specified in the comments within the FlaskXSS source code. First, it is checked whether a data object originates from an operation that obtains data externally. Examples of such operations are database queries or the retrieval of request data. In this case, the `OUT` decorator is added to the tag, followed by the `::` delimiter. The other considered case is that a data object is passed as a function argument, represented by the `IN` decorator. Furthermore, since functions can receive multiple function arguments, the name of the argument that the data object is passed to is stated, followed by the `::` delimiter. Next, the fully qualified name of an operation has to be set. This refers to the specification of function or class attribute names for the declaration of source and sink models,

illustrated in Chapter 3.1.3. The fully qualified names will be used during the generation of security rules; hence they must be in a format that Pysa can interpret. In any case, the key of a tag is composed of the respective directional decorator and the operation's fully qualified name. If a tag represents an `IN` operation, its value consists of the name of a function argument, followed by the `::` delimiter and the value of the data object. If a tag represents an `OUT` operation, the tag-value consists of the value of the data object. Adhering to the described semantics and syntax, spans were instrumented for the FlaskXSS HTTP handling logic, contained in the functions of `flaskxss.py`. Listing 16 is used to illustrate an example of the instrumentation of each OUT and IN operation tags. It is taken from the persistent HTTP handler function from FlaskXSS.

```
1 span.set_tag('OUT::werkzeug.wrappers.BaseRequest.form',
2              str(request.form['comment']))
3 comment = request.form['comment']
4
5 span.set_tag('IN::def db.add_comment(comment)',
6              ('comment::' + str(comment)))
7 db.add_comment(comment)
```

Listing 16 Example for the instrumentation of OUT and IN tags

Consequently, the traces have to be sent to a tracing backend. Tracers instantiated by the `init_tracer` function within `tracing.py` automatically report terminated spans to the running and listening Jaeger collector. The collector is included in the "all-in-one" Jaeger executable, which launches all services necessary to collect, display and extract Jaeger traces. It can be deployed using the pre-built Docker image available on DockerHub. As part of the prototype, the `docker-compose.yml` file was written, containing all the instructions necessary to start a Docker container running the "all-in-one" Jaeger image. Starting the image exposes several ports that the Jaeger backend listens to, in order to collect traces or serve other functionalities [The21c]. Relevant for the remaining part of this thesis is port `16686` which serves the Jaeger front-end as well as the Jaeger Query Service; an HTTP JSON API for the retrieval of traces [The21a].

4.2 Active Scanning Process

After implementing the tracing instrumentation, the active scanning process of ZAP must be integrated to generate informative traces as well as to obtain the results of the DAST analysis. All the logic required to run an active scan as well as its related process steps, which are described in Chapter 3.1.2, is included in the `ZAP_scan.py` file of the `iast-prototype/app/ZAP` subdirectory. Here, two functions, namely

crawl and activeScan, are defined for the use of the traditional spider and the deployment of the active scan rules, based on the official ZAP API documentation [ZAP21g]. To enable communication between the prototype and OWASP ZAP, first, a ZAP client instance is created by specifying the API key, taken from the settings menu of the ZAP Desktop application. The key is defined as a static variable within the ZAP_Config configuration class of the config.py file.

The crawl function expects an argument target, which defines the root URL of the web application to be scanned. When executed, it proxies an initial request using the urlopen function and then proceeds to deploy the traditional spider onto the target with the zap.spider.scan(target) function call. This function returns an ID which can be referenced to extract information of a specific crawling process. The progress of the spider is polled throughout the process and once the spider status reaches 100, a list of all identified URLs is retrieved by calling the zap.spider.results(scanID) function call and returned as crawl_results variable.

Similar to the crawl function, the activeScan function expects the same target argument. It generates a scanID by running the active scanner against the given target URL, calling the zap.ascan.scan(target) function. Once all active scan rules have been deployed and the active scanning status returns 100, the alerts are retrieved using the zap.core.alerts function. In the context of this function call, the argument baseurl is set to the target's root URL and the argument riskId is set to 3. Thus, only alerts that originate from one of the application's URLs and indicate a vulnerability with a high risk level are retrieved. Since XSS alerts correspond to a risk level three, various non-XSS alerts are already sorted out by this function call. The returned variable active_scan_results is of datatype list and contains a set of alert dictionaries.

4.3 Filtering ZAP Alerts

Based on the list of dictionaries obtained through running the activeScan function, the ZAP_filterResults.py file was created within the app/ZAP directory. Two functions named extractXSSAlerts and extractRelevantInfo are included in this file. The task of the functions is to isolate sets of specific data from a list of relevant XSS alerts. The structure of the examined list of dictionaries, retrieved through the activeScan function call can be taken from the section "Getting the Results" of the ZAP API documentation [ZAP21c].

The extractXSSAlerts function expects the argument dicList, a list that contains dictionaries consisting of all previously queried alert dictionaries, and the list argument alerts, containing string identifiers for the alerts that are to be extracted from dicList. The function then loops through each key-value pair of each dictionary within dicList. If the value of a dictionary-entry with the "alert" key corresponds to a value defined in alerts, the entire dictionary is added to a list. This process repeats itself until all dictionaries have been checked and the resulting list of dictionaries is stored in the variable xss_alerts and returned by the function. The alerts argument is defined as ALERT_FILTER_VALUES within the ZAP_Config configuration class and contains the values "Cross Site Scripting (Reflected)" and "Cross Site Scripting (Persistent)". Thus, through the deployment of this function, only reflected and persistent XSS alerts are retrieved.

The extractRelevantInfo function expects the argument dicList, a list of dictionaries containing the previously filtered list of XSS alerts, and the argument keys, being a list of key strings that dicList should be filtered for. Comparable to the extractXSSAlerts function, it loops through dicList eliminating all entries that are not specified in keys by adding every matching key-value pair to a new list of dictionaries. This list of dictionaries is then returned by the function. The keys argument is defined as KEY_FILTER_VALUES within the ZAP_Config configuration class and contains the values "name, url, method, param, attack". Thus, from a list of dictionaries containing the reflected and persistent XSS alerts of an active scanning process, only the key-value pairs matching keys are extracted. At this point, all information, necessary for the remaining IAST operations is extracted from the penetration testing process of OWASP ZAP. For this reason, the resulting list of dictionaries is written to a newly created .json file within the output folder. The title of the JSON file is provided with a timestamp of the execution. Furthermore, the JSON file ZAP_report_example.json was included to serve as reference for such a report.

4.4 Retrieval and Filtering of Traces

During the execution of the activeScan function, due to the tracing instrumentation, traces are transferred to the Jaeger Collector and accumulated within the storage backend [The21b]. The high quantity of active scan requests proxied through ZAP generates a large amount of tracing data. The Jaeger_extract.py file within the app/Jaeger directory contains two functions designed to extract information that is useful for the IAST process from the vast amount of collected tracing data. The function getTraces retrieves tracing data via the HTTP JSON Jaeger Query

Service API, and the function `getSpans` filters them according to the information from ZAP. It should be noted that the API endpoint used for the `getTraces` function is undocumented and prone to changes [The21a]. It is primarily used to retrieve traces for the Jaeger GUI, hence the API's definition can be derived by operating inside the GUI and inspecting the requests sent to the Jaeger Query Service. An unofficial specification can be taken from this GitHub issue-thread: [bla18].

The `getTraces` function expects the arguments `url` and `method`. In the context of the IAST process the `url` argument is a `string`, taken from the "url" value of an XSS alert. Similarly, the `method` argument corresponds to a `string` containing the alert's "methond" value. With the use of these arguments and additional variables imported from the `Jaeger_Config` configuration class, a URL directed at the Jaeger-internal HTTP JSON Jaeger Query Service API is composed in order to retrieve traces [The21a]. The URL consists of the variables `base`, `service`, `limit`, `operation` and `method`. The `base` variable is imported from the `Jaeger_Config.JSON_API` class variable and contains the root URL of the API. The `service`-variable specifies the name of the application that was instrumented with the tracing system and is set to "FlaskXSS" within the `SERVICE` configuration class variable. The `limit integer` variable is also taken from the configuration class, where the class variable `LIMIT_TRACES` was set to 1000 in order to ensure that all traces generated through ZAP are taken into account. The `operation`-, and `method`-variable are generated through the received function arguments `url`, and `method`. The `url` argument contains the request-URL that ZAP utilized to execute an XSS exploit on a specific http handler function. Referring back to the instrumentation of traces, request URLs are contained in all traces and thus can be taken into account when retrieving traces. The same applies for the `method` argument which contains the request-method deployed by ZAP for an XSS exploit. Through the targeted query of traces that contain specific metadata the retrieval is narrowed down by embedding the described variables in the API query `string`. The exemplary composition of a query `string` is depicted within the `getTraces` comments. Consequently, the query is submitted and the results are returned as a `list` of `dictionaries`.

The `getSpans` function expects the arguments `lisdic` and `attack`. The `lisdic` argument corresponds to the list of traces returned by the `getTraces` function. The `attack` argument contains the attack-value of one XSS alert retrieved from the list of alerts from the `ZAP.extractRelevantInfo` function. The function thus loops through the nested structure of the `lisdic` tracing data and examines each value of a tag within each tag of a span within each span of a trace. Once a value of a tag is detected that contains the value of `attack`, the entire span is

added to the list-variable `span_out` and the loop terminates. There is no use in finding several spans corresponding to one alert. Thus, one span that contains the attack payload of one ZAP alert is extracted from the `lisdic` list of traces and returned by the function. This reveals the metadata of all operations that are related to a successful XSS exploit by ZAP. Since all tracing data necessary for the IAST process has been collected, the spans extracted through the `getSpans` function are stored in the `app/output` folder within a `.json` file. The title of the file is provided with a timestamp of the execution. Moreover, the example output file `Jaeger_traces_example.json` is included in the `output` folder as reference.

4.5 Dynamic Generation of Security Rules

The documented functions above implement the creation of supplementary contextual data in the form of operations and related data values traced during the exploit of an XSS vulnerability. The return value of the `getSpans` function contains all information necessary to generate security rules for the execution of Pysa. For this, the functions `extractSourcesSinks`, `writeSource`, `writeSink` and `readResults` are defined within the `app/Pysa/Pysa_generateRules.py` file. The functions handle the extraction of sources and sinks, the writing of security rules, and the output of the security rules in a presentable format. It is evident that the successful creation of sources and sinks is primarily dependent on the metadata found in tags. For this reason, the instrumentation of traces is first brought into the context of the generation of Pysa security rules. Consequently, the named functions are explained.

In compliance with the instrumentation logic explained in Chapter 4.1, primarily operations and the corresponding data objects were included a span's tags. A distinction between operations that retrieve data externally and functions that operate on such data has been made. The operations thus were labeled with the respective decorators `OUT`, and `IN`. In the context of Pysa security rules, an operation marked with the `OUT` decorator corresponds to a potential taint source. An operation labeled with the `IN` decorator represents a potential sink. A directional decorator is followed by the fully qualified name of an operation, separated by the `::` delimiter. A tag's declaration of fully qualified names is in accordance with the syntax Pysa adheres to in order to recognise taint flows. Hence, during the dynamic generation of security rules, operation names can be adopted without applying significant changes. Furthermore, since malicious data can lead to the exploitation of a vulnerability after it has been modified by an operation, a general distinction between *exact* and *implicit* sources and sinks was made. The data object of an exact source or sink contains the exact value of the attack payload. The extrac-

tion of the request.form['comment'] class attribute, derived from the FlaskXSS /persistent HTTP handler function serves as an example for an exact source. Implicit sources and sinks comprise operations that derive or process data objects that contain other data values in addition to the attack payload string. As an example of an implicit source, the db.get_comments() function of the /persistent HTTP handler returns a list of data values that includes the attack payload apart from other values retrieved from the FlaskXSS database.

The extractSourcesSinks function expects the argument lisTags, a list of tags (span) that originates from the getSpans function, and the argument attack, being an attack payload derived from one ZAP XSS alert. The function detects the source and the sink of an attack payload by sequentially examining the data object of every tag in lisTags. For this, the operations correlated to the data object that contains the value of attack are appended to lists. Whenever a source operation is detected, it is appended to a source-list. If a sink operation is found, its operation is stored in a sink-operation-list, and its argument is stored in a sink-argument-list. The detection of source and sink operations is carried out under consideration of the distinction between exact and implicit sources and sinks. Thus, six lists can be populated in total. After the entire span has been examined, the populated lists represent a chronological order of all potential sources and sinks. Since only one true source can exist for each exact and implicit source, the first element of both lists is retrieved. It corresponds to the operation responsible for the first appearance of the attack payload within a span. A similar approach is taken for the retrieval of sinks. Here, however, the last element is extracted from both exact and implicit sink-operation- and sink-argument-lists. Through this, operation and argument in which a tainted value terminates are retrieved. Consequently, the extractSourcesSinks function extracts the information needed to write Pysa source and sink models from a list of tags. Exact and implicit sources and sinks are ultimately passed to the writeSource and writeSink functions.

The writeSource function expects the argument source, being a source operation string, originating from the extractSourcesSinks function, and the argument rule. As explained in Chapter 3.1.3, Pysa recognizes taint flows by correlating source, sink and rule definitions derived from a taint.config file, and models specified in various .pysa files. The rule argument corresponds to an XSS TaintSource[SOURCE_NAME] decorator and is derived from the Pyre_Config configuration class of the config.py file. By concatenating source and rule, a string is created which matches the type annotation syntax for models and can thus be interpreted by Pysa. Ultimately, this model is written to the sources_sinks.pysa

file contained in the app/Pysa directory. Similarly, the writeSink function concatenates the string arguments it receives in order to create a sink model. However, function sinks are composed of the operation's name, argument and a taint.config TaintSink[SINK_NAME] decorator. For this reason, the writeSink function expects the arguments sink_op, sink_arg and rule. The resulting model is written to the same sources_sinks.pysa model file as the writeSource function.

After all models have been written to the sources_sinks.pysa model file, the readResults function can be deployed to extract the newly generated models. For this purpose, each model is added to a list of strings which is then presented in the GUI of the IAST prototype.

4.6 Running the IAST Prototype

After a detailed description of the implemented features for the IAST prototype has been provided, this section serves to gradually illustrate the execution of all related process steps and puts them into perspective. The IAST process comprises the initialisation of all required services, the execution of automated penetration testing, the subsequent generation of security rules, and the concluding taint analysis. The Pysa taint analysis is conducted in two variants. Firstly, it is carried out only using the dynamically generated model file. Secondly, taint analysis is performed using both dynamically generated models, and the pre-defined models declared in the flask_sources_sinks.pysa file, which was used in Chapter 3.1.3. In conclusion, the results of both executions are displayed in the end of this section. The specifications of the machine on which this run was conducted and the majority of the development were realised is depicted in Table 7 in the appendix. To ensure reproducibility, the installation of all necessary services, as well as the setup of the IAST prototype, are explained in the README.md file within the project directory.

Starting the prototype architecture's services is not within the IAST prototype's scope of functions; they must be started manually. Thus, as a precondition for the deployment of the IAST process, the services FlaskXSS, Jaeger, and the IAST prototype itself are launched within their respective terminal sessions. In addition, the OWASP ZAP Desktop application is started. Furthermore, it should be noted that repeated active scanning leads to large amounts of malicious data stored in the FlaskXSS database, which will influence ZAP's results due to changes to the DOM. Therefore, the FlaskXSS database is flushed before each execution of the IAST process. Regarding the concluding execution of Pysa, the pre-defined taint.config file of the Pyre-Check project directory is transferred to the app/Pysa directory.

In the context of the first taint analysis run, adjustments to the global configuration file of Pysa are applied so that it refers to the dynamically generated `.pysa` model file and the copied `taint.config` file within the `app/Pysa` directory. For the second execution of taint analysis, the configuration file is modified to also include the `flask_sources_sinks.pysa` file in the `assets_thesis/pysa_security_rules` directory. The exact execution steps of the mentioned preconditions above are comprehensively described in the `README.md` file.

Upon calling the prototype's URL, the IAST prototype dashboard is displayed. Its main components are a form containing a single text field that expects and validates the URL of the target application, as well as a submit button that starts the IAST process. Hence, the FlaskXSS target root URL is specified and confirmed, sending a `POST` request to the underlying HTTP handler. In the course of the started process, information about the status of the prototype can be obtained from the respective terminal. The target application's URL is passed within the handler function to the previously documented functions, starting with the `crawl` function. After all URLs within the scope of FlaskXSS are identified, the list of web crawling results is returned to the handler function. The target URL is then forwarded to the `activeScan` function, which proxies active scan requests to the preciously identified URLs stored in the current ZAP session. For the purpose of comprehensibility, the requests generated in this process step can be taken from the `ZAP_ascan_requests_reference.csv` file within the `assets_thesis` directory. After completion of the active scanning process, the extracted list of alerts is passed to the `extractXSSAlerts` function. It is also given a list of alert filter values as a function argument. By providing these variables, a list containing only reflected and persistent XSS alerts is returned. Next, the list of reflected and persistent XSS alerts is given to the `extractRelevantInfo` function, together with a set of key filter values. This returns the passed list of alerts containing only the relevant alert data values "name", "url", "method", "param" and "attack". With the thereby obtained "ZAP report", the DAST subprocess ends, and the extraction of simultaneously generated traces is started. For this, the `getTraces` function first queries a selection of traces from the Jaeger API. The list of traces thereby obtained is passed to the `getSpans` function together with the information of a ZAP alert. This extracts the alert's span, meaning the sequence of certain operations that represent the given alert. Subsequently, the `extractSourcesSinks` function is executed by passing this span and the attack payload of the corresponding alert. Thereby, the sources and sinks of the attack payload are retrieved and written to the `sources_sinks.pysa` file of the `app/Pysa` directory. This process is individually repeated for all XSS alerts extracted through the DAST subprocess. Thus, Pysa source and sink models are

declared for the XSS vulnerabilities found in the “ZAP report”. At this point, the IAST prototype terminates, and the outcomes of web crawling, active scanning, and the generation of security rules are displayed in the IAST dashboard. The resulting dashboard display is presented in Figure 3.

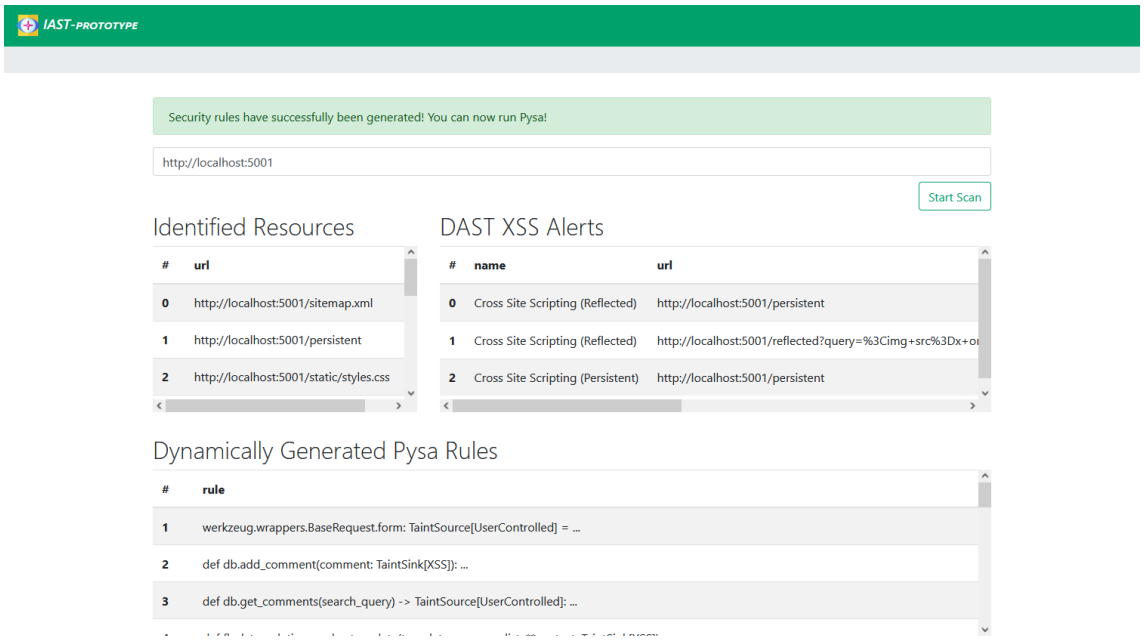

Figure 3 IAST prototype dashboard results

In the final step of the IAST process, taint analysis is performed by Pysa. The results of the first taint analysis run under consideration of the dynamically generated models are shown in Table 4.

Name	Code	Location	Line	Description
XSS	5008	flaskxss.insufsanitize	141	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.persistent	72	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.persistent	81	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.reflected	108	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.reflected	108	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"

Table 4 IAST results using dynamically generated models

The second run uses the pre-defined source and sink model declarations of the `flask_sources_sinks.pysa` file in addition to the dynamically generated models. The results of this taint analysis run are displayed in Table 5.

Name	Code	Location	Line	Description
XSS	5008	flaskxss.insufsanitize	141	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.persistent	72	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.persistent	81	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.reflected	108	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.reflected	108	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
XSS	5008	flaskxss.sufsanitize	170	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"
Potential Server Side Template Injection	6073	flaskxss.sufsanitize	170	"Potential Server Side Template Injection [6073]: User-controlled data may eventually flow into a Server Side Template Injection sink"
XSS	5008	flaskxss.xssheader	196	"XSS [5008]: Data from [UserControlled] source(s) may reach [XSS] sink(s)"

Table 5 IAST results using both pre-defined and dynamically generated models

5 Evaluation of the IAST Prototype

The previous chapter covered the implementation of the IAST prototype features based on the architecture conceptualised in Chapter 3.3. It provides a precise explanation of all relevant components, their functionalities as well as their interaction within the IAST process. Ultimately, the execution of the prototype was presented. The next step is to analyse the resulting sets of output and to understand their technical backgrounds. This provides the basis for the subsequent critical assessment of the obtained results as well as the IAST prototype in its entirety.

5.1 Deduction of Results

Since the results of the IAST prototype are based on the intermediate results of its individual subprocesses, they need to be analysed first. A distinction is made between the results of the automated penetration testing, the dynamic declaration of source and sink models, and lastly, the taint analysis.

The starting point of the IAST prototype is the execution of automated penetration testing. For this purpose, OWASP ZAP was integrated in the IAST process. Through its deployment, the reflected and persistent XSS vulnerabilities within the `/persistent` view, as well as the reflected XSS vulnerability within the `/reflected` view were returned. The results and their associated process steps are given in Chapter 3.1.2.

Subsequently, source and sink models are declared based on the occurrences of certain XSS attack payloads within a span. As outlined in Chapter 4.5, the attack payload information of XSS alerts is utilized for the identification of sources and sinks. Then, the metadata of the corresponding tags is extracted to write source and sink models. The files `Jaeger_traces_example.json` and `ZAP_report_example.json` within the `app/output` directory can be referred to in order to reconstruct the declaration of source and sink models. They represent a valid pair of XSS alerts and corresponding spans that lead to the results of this subprocess, displayed in Table 6.

In line with the statements of Chapter 3.1.1, an XSS taint flow within the `persistent` HTTP handler function is recognized. It is associated with the retrieval of untrusted form data followed by its storage within the underlying database. This constitutes a persistent XSS vulnerability. The attack payload of this taint flow originates from the `BaseRequest.form` class attribute, thus explaining the source model declaration as user-controlled data. The exact payload was last detected in

route	source declaration	sink declaration
/persistent	werkzeug.wrappers.BaseRequest.form: TaintSource[UserControlled] = ...	def db.add_comment (comment: TaintSink[XSS]): ...
/persistent /reflected	def db.get_comments(search_query) -> TaintSource[UserControlled]: ...	def flask.templating.render_template (template_name_or_list, **context: TaintSink[XSS]): ...
/reflected	werkzeug.wrappers.BaseRequest.args: TaintSource[UserControlled] = ...	def flask.templating.render_template (template_name_or_list, **context: TaintSink[XSS]): ...

Table 6 Dynamic model declaration results

the `db.add_comment(comment)` function argument, suggesting that the taint flow terminated in this operation. Hence, the `comment` argument of the `db.add_comment` function was declared as the taint sink of the `BaseRequest.form` taint source.

Another XSS taint flow was detected in the dynamic inclusion of untrusted database values within the returned view of the `persistent` handler. ZAP has reported this as a reflected XSS vulnerability that is connected to the previously outlined persistent XSS vulnerability. The IAST prototype identified a modified version of the previous attack payload, retrieved through the `db.get_comments()` function call. In this context, the function returned a `list` of all `comment` entries of the `comments` table. The previously stored XSS attack payload was included in one of these entries, which is why this function was detected as a taint source. This implicitly malicious data value was last observed in the `**context` argument of the `render_template` function. The reflected XSS taint flow thus terminates, explaining the corresponding XSS sink declaration.

In accordance to the description of the `reflected` HTTP handler function in Chapter 3.1.1, the taint flow of its reflected XSS vulnerability was identified. It is caused by the retrieval of untrusted query `string` data and the reflection of this data within the generated view. The attack payload of the taint flow was retrieved through the `BaseRequest.args` class attribute, resulting in the user-controlled source model declaration. The exact payload ultimately terminated in the `**context` argument of the `render_template` function, justifying the corresponding XSS sink model declaration.

Within the same handler function, the XSS taint flow caused by the retrieval of malicious database values was detected. It is related to the persistent XSS vulnerability of the `persistent` HTTP handler. The `get_comments` and `render_template` function calls constitute this taint flow which are also used in the `reflected` HTTP handler.

Lastly, the results of the taint analysis are analysed. The first run of Pysa was performed using the populated model file from the previous subprocess. The predefined `taint.config` rule definitions associate XSS taint flows originating from `UserControlled` sources that terminate in either `XSS` or `StringMayBeHTML` sinks. Combining the information of both files, the results from Table 4 were obtained.

Two taint flows were detected within the `persistent` handler function. One arises from the `request.form` class attribute source and terminates in the function argument sink `add_comment(comment)` in line 73. This is followed by the `get_comments` function call source which terminates in the `render_template(**context)` function argument sink.

The two taint flows detected within the `reflected` handler function both terminate in the `render_template(**context)` function argument sink of line 113. One of the two listed taint flows originates from the `request.args` class attribute source. The alternative taint flow originates from the the `db.get_comments` function call source.

Furthermore, through the declaration of source and sink models, a taint flow was identified that is not explicitly indicated in the results of ZAP. The `insufsanitize` handler function's, taint flow between the `request.args` class attribute source and the `render_template(**context)` function argument sink was detected. This is based on the previous declarations of source and sink models.

All taint flows that were not detected during the first run are due to missing model declarations. For this reason, the predefined `flask_sources_sinks.pysa` model file was added as a supplement. Examining the results shown in Table 5, it is noticeable that the results of the first run have been extended by the results depicted in Chapter 3.1.3. A respective interpretation of the additional results is given in Chapter 3.2.

5.2 Critical Appraisal

The goal of the implementation of the IAST prototype was to improve AST results through the combination of DAST and SAST methods. Based on the concept established in Chapter 3.3, it was assumed that all XSS vulnerabilities of FlaskXSS thus would be detected. In the following, a critical analysis of the development of the IAST prototype, its functionality, and the prototype in its entirety is conducted.

First, a comparison between the results of the central use case, presented in Chapter 3.1, and the prototype is drawn. In this context, the outcome of the use case, being

the results achieved by the individual deployment of ZAP and Pysa, are considered in their entirety. This yields a total of six different detected issues. Through the interactive approach of the IAST process, a maximum of eight different results was returned. Thus, an improvement in the quantity of results is given. Furthermore, the quality of results has increased. In total, four true positive vulnerabilities were reported in the context of the use case. The IAST prototype identified a maximum of seven true positive vulnerabilities in FlaskXSS. The amount of false positive results has not increased compared to the central use case where one false positive was detected. Moreover, the assumption made in Chapter 3.3 was confirmed as all vulnerabilities of FlaskXSS were detected by the IAST prototype. Based on the results, it is evident that the objective of improving AST results has been achieved.

Nevertheless, the concept that the IAST prototype represents should be critically evaluated according to the theoretical findings gathered in Chapters two and three. The theoretical foundations of IAST are presented in Chapter 2.2.3. This prototype reflects the described characteristics of an IAST tool as methods of both DAST and SAST were successfully combined. Through the instrumentation of "agents", runtime context is incorporated in the analysis of code. The resulting monitoring of control and data flow provides the prototype with more contextual information, justifying the improvement of results compared to DAST and SAST. However, limitations have been perceived in the research for IAST. The term IAST is a relatively new and imprecise term. When researching the implementation of IAST tools, it appears that the term does not provide an exact conceptual framework. Literature documenting the techniques employed in IAST tools is scarce, and commercial IAST products do not provide any insight into how they operate. The characteristics of IAST tools presented in Chapter 2.2.3, therefore, are common properties derived from a limited availability of literature. The prototype and its concept are less derived from an understanding of IAST. Instead, it was logically deduced from an understanding of DAST and SAST. Hence, it is questionable whether the IAST prototype is an IAST approach at all. Moreover, the framework of the central use case presented in Chapter 3.1 must be critically considered. For demonstration purposes, the vulnerabilities of FlaskXSS are very simplistic. Most web frameworks, including Flask, contain security mechanisms that prevent the represented XSS vulnerabilities. For the use case, the `autoescape` function of Jinja was disabled. Regarding the functionalities of FlaskXSS this is not a realistic setting. Furthermore, Chapter 3.1 described the deployment of ZAP and Pysa with minimal manual configuration effort. It can be assumed that with the proper configuration of both tools, better results are achieved. Overall, the central use case deviates strongly from a realistic setting. Thus, the question arises whether the concept of the IAST prototype can be

applied in reality. However, the task of a prototype is to exemplify a concept within a simplified but functional product. The IAST prototype fulfills this purpose.

Finally, limitations that arose during development are discussed. The integration of tracing posed difficulties in finding the optimal Python tracing module. Ideally, traces should be instrumented automatically. Since such a tracing module does not exist, and the development of auto-instrumentation was outside the scope of this thesis, traces were instrumented manually. In addition, an unstable and undocumented Jaeger API was adopted for the retrieval of tracing data. According to the Jaeger API documentation, the recommended way of retrieving traces is via its gRPC endpoint [The21a]. Due to time constraints, it was decided against such an implementation. During the development of the `extractSourcesSinks` function, limitations were recognized that are related to the instrumentation of tags. According to the Pysa documentation, in addition to class attributes and function arguments, entire classes can be declared as source or sink [Fac21a]. Since such a taint source or sink was not included in FlaskXSS, it was neither considered in the instrumentation of tags nor the logic of the `extractSourcesSinks` function. For the same reason, only function sinks are taken into account for model declaration. In addition, according to the Pysa `taint.config` rule definitions, XSS taint flows are composed of `UserControlled` sources and `XSS` or `StringMayBeHTML` sinks. The prototype does not differentiate between different types of sinks. Moreover, the distinction between exact and implicit sources and sinks, made in Chapter 4.5, must be reviewed critically. In consideration of an XSS alert, precisely one exact pair of source and sink can exist within an HTTP handler function. However, an infinite number of implicit source and sink pairs can occur. The `extractSourcesSinks` function only distinguishes between taint flows that propagate *exactly* the alert's attack payload and taint flows that *partially* propagate the alert's attack payload. Optimally, the function would compare all data values of a span with every tainted value it can identify based on an XSS alert. Thus, an identification of all possible taint flows would be given. Instead, the `extractSourcesSinks` function is only able to identify a maximum of two taint flows. The reason why a comparison based on implicitly tainted values has not been implemented is that Jaeger only allows a limited length of tag value `strings`. As soon as a value `string` reaches a certain length, e.g., when it contains the results of a large database query, parts of the `string` are cut off. This makes a comparison based on tag values impossible. More information about this limitation can be taken from the comments of this function.

6 Conclusion and Outlook

The contents of this thesis represent the exemplary implementation of a novel AST approach for Python web applications in the face of XSS vulnerabilities. In the course of this, the theoretical foundations of XSS were covered and brought into the context of AST. A fundamental understanding of the AST categories DAST and SAST as well as their role in the prevention of XSS was conveyed. Moreover, the concept of IAST and its characteristics were highlighted. Based on this theoretical background, the objective of finding a concept for the prototypical development of an IAST tool was dealt with. For this purpose, a central use case demonstrating the analysis of an application with XSS vulnerabilities through the individual deployment of SAST and DAST tools was presented. This setting was intended to demonstrate the open source AST tools OWASP ZAP and Pysa, as well as to identify requirements for the IAST prototype based on their individual limitations. Through this, an architecture that incorporates the functionalities of both tools was developed and implemented. Consequently, the IAST prototype was tested, and findings were drawn from its results.

The research question "How can methods of SAST and DAST be combined in the form of IAST?" was answered based on the contents of this bachelor thesis. Furthermore, the results of the IAST prototype indicate the concept's potential. Its decisive advantage lies in the generation of contextual information by connecting data that stems from inside an application and data that stems from outside. The analysis of control and data flow under consideration of runtime context offers a significant perspective for further development. For example, the declaration of sanitizer models could be achieved by detecting the interruption of taint flows. In addition, a feature that recognizes congruent results of Pysa and ZAP could be implemented in order to indicate the prototype's confidence in a vulnerability. Possibilities for enhancement also arise from the technical limitations of the prototype. In this context, the automatic instrumentation of tracing is of particular relevance.

Web applications have already proven to be an indispensable part of most people's daily lives. However, with the increasing relevance of applications offered on the Internet, the negative impact of vulnerabilities in web applications increases as well. Furthermore, due to continuously evolving functionalities, the detection of such vulnerabilities is steadily growing in complexity. For this reason, the development of concepts for the automated prevention of security flaws, as presented in this thesis, will remain of significance.

Appendix

Processor	AMD Ryzen 5 2600 Six-Code Processor 3.40 GHz
Memory	16 GB DDR4 - 3200MHz
Operating system	Windows 10 Education Version 1909
System type	64-bit Operating System, x64-based processor

Table 7 Computer specifications

References

[Acu14] Acunetix. *Cross-Site Scripting in HTTP Headers*. 2014. URL: https://www.acunetix.com/blog/articles/xss-http-headers/ (visited on 09/20/2021).

[And10] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Hoboken: John Wiley & Sons Inc, 2010.

[bla18] black-adder. *Jaeger: Issue #456 - Document external interfaces*. Ed. by GitHub. 2018. URL: https://github.com/jaegertracing/jaeger/issues/456 (visited on 09/21/2021).

[Cro+21] Roland Croft et al. *An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing*. Ed. by Association for Computing Machinery. Italy, 2021. URL: http://arxiv.org/pdf/2107.01921v2.

[CWE06] CWE. *CWE-79: Improper Neutralization of Input During Web Page Generation*. Ed. by MITRE. 2006. URL: https://cwe.mitre.org/data/definitions/79.html (visited on 09/20/2021).

[EK13] Patrick Henry Engebretson and David Kennedy. *The basics of hacking and penetration testing: Ethical hacking and penetration testing made easy*. 2nd ed. The basic. Amsterdam: Syngress/Elsevier, 2013.

[Fac21a] Facebook Open Source. *Pyre-Check Documentation: Getting Started*. 2021. URL: https://pyre-check.org/docs/pysa-basics/ (visited on 09/21/2021).

[Fac21b] Facebook Open Source. *Pyre-Check Documentation: Installation*. 2021. URL: https://pyre-check.org/docs/installation (visited on 09/21/2021).

[Fac21c] Facebook Open Source. *Pyre-Check Documentation: Quickstart*. 2021. URL: https://pyre-check.org/docs/pysa-quickstart/ (visited on 09/21/2021).

[Ghi20] Devndra Ghimire. "Comparative study on Python web frameworks: Flask and Django". Bachelorarbeit. Metropolia University of Applied Sciences, 2020. URL: https://www.theseus.fi/bitstream/handle/10024/339796/Ghimire_Devndra.pdf?sequence=2&isAllowed=y (visited on 09/20/2021).

[Her19] Michael Herman. *Django vs. Flask in 2021: Which Framework to Choose*. Ed. by testdriven.io. 2019. URL: https://testdriven.io/blog/django-vs-flask/ (visited on 09/20/2021).

[Hög20] Jonas Höglund. "An Analysis of a Distributed Tracing Systems Effect on Performance: Jaeger and OpenTracing API". PhD thesis. Umea University, 2020. URL: https://www.diva-portal.org/smash/get/diva2:1475588/FULLTEXT01.pdf (visited on 09/21/2021).

[HN17] F. Holik and S. Neradova. "Vulnerabilities of modern web applications". In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Ed. by Petar Biljanovic. Piscataway, NJ: IEEE, 2017, pp. 1256–1261. DOI: 10.23919/MIPRO.2017.7973616.

[Hup20] Thomas Hupperich. *Introduction to IT Security - Web Security II: Cross-Site Scripting*. Münster, 2020.

[Hup19] Thomas Hupperich. *IT Security in Digital Business - Perspective of IT Security*. Münster, 2019.

[Kir21] KirstenS. *Cross Site Scripting (XSS)*. 2021. URL: https://owasp.org/www-community/attacks/xss/ (visited on 09/20/2021).

[Kvo20] Elena Kvochko. "Why Cyber Security Is Still So Complex". In: *Forbes* (2020). URL: https://www.forbes.com/sites/elenakvochko/2020/10/25/why-cyber-security-is-still-so-complex/ (visited on 09/20/2021).

[Lin21] Jessica Lin. *ELI5: Pysa - A Security-Focused Static Analysis Tool for Python Code*. Ed. by Facebook for Developers. 2021. URL: https://developers.facebook.com/blog/post/2021/04/29/eli5-pysa-security-focused-analysis-tool-python/?locale=de_DE (visited on 09/21/2021).

[Mak21] Matt Makai. *flask.templating render_template_string Example Code*. Ed. by Full Stack Python. 2021. URL: https://www.fullstackpython.com/flask-templating-render-template-string-examples.html (visited on 09/20/2021).

[MK15] Yuma Makino and Vitaly Klyuev. "Evaluation of web vulnerability scanners". In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Piscataway, NJ: IEEE, 2015, pp. 399–402. DOI: 10.1109/IDAACS.2015.7340766.

[MD19] Achmad Fahrurrozi Maskur and Yudistira Dwi Wardhana Asnar. "Static Code Analysis Tools with the Taint Analysis Method for Detecting Web Application Vulnerability". In: *2019 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2019, pp. 1–6. DOI: 10.1109/ICoDSE48700.2019.9092614.

[Mat+20] Francesc Mateo Tudela et al. "On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications". In: *Applied Sciences* 10.24 (2020), p. 9119. DOI: 10.3390/app10249119.

[MHM19] Vicente de Mohino, Bermejo Higuera, and Sicilia Montalvo. "The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies". In: *Electronics* 8.11 (2019), p. 1218. DOI: 10.3390/electronics8111218.

[Moz21] Mozilla. *Referer*. Ed. by MDN Web Docs. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer (visited on 09/20/2021).

[Ope21a] OpenTracing. *OpenTracing API Documentation: Overview*. 2021. URL: https://opentracing.io/docs/overview/ (visited on 09/21/2021).

[Ope21b] OpenTracing. *OpenTracing API Documentation: What is Distributed Tracing?* 2021. URL: https://opentracing.io/docs/overview/what-is-tracing/ (visited on 09/21/2021).

[OWA21] OWASP Foundation. *Source Code Analysis Tools*. 2021. URL: https://owasp.org/www-community/Source_Code_Analysis_Tools (visited on 09/20/2021).

[Pal10a] Pallets. *Flask Documentation: API*. 2010. URL: https://flask.palletsprojects.com/en/2.0.x/api/ (visited on 09/20/2021).

[Pal10b] Pallets. *Flask Documentation: Foreword*. 2010. URL: https://flask.palletsprojects.com/en/2.0.x/foreword/ (visited on 09/20/2021).

[Pal10c] Pallets. *Flask Documentation: Foreword for Experienced Programmers*. 2010. URL: https://flask.palletsprojects.com/en/2.0.x/advanced_foreword/ (visited on 09/20/2021).

[Pan19] Yuanyuan Pan. "Interactive Application Security Testing". In: *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*. IEEE, 2019, pp. 558–561. DOI: 10.1109/ICSGEA.2019.00131.

[PT 19] PT Security. *SAST, DAST, IAST, and RASP: how to choose?* Ed. by Positive Technologies. 2019. URL: https://www.ptsecurity.com/ww-en/analytics/knowledge-base/sast-dast-iast-and-rasp-how-to-choose/# (visited on 09/20/2021).

[PyT20] PyT. *PyT: A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications*. Ed. by GitHub. 2020. URL: https://github.com/python-security/pyt (visited on 09/21/2021).

[Pyt21] Python Software Foundation. *Python 3.9.7 documentation: HyperText Markup Language support*. 2021. URL: https://docs.python.org/3/library/html.html (visited on 09/20/2021).

[Roh19] Matthias Rohr. *Continuous Security Testing mit IAST*. Ed. by OWASP Foundation. 2019. URL: https://owasp.org/www-pdf-archive/Continuous_Security_Testing_mit_IAST.pdf (visited on 07/04/2021).

[Sag+18] Deepika Sagar et al. "Studying Open Source Vulnerability Scanners for Vulnerabilities in Web Applications". In: *IIOAB Journal Vol. 9* (2018), pp. 43–49. URL: https://www.iioab.org/IIOABJ_9.2_43-49.pdf (visited on 09/21/2021).

[Sca18] Thomas Scanlon. *10 Types of Application Security Testing Tools: When and How to Use Them*. 2018. URL: https://insights.sei.cmu.edu/blog/10-types-of-application-security-testing-tools-when-and-how-to-use-them/ (visited on 09/20/2021).

[SEB20] Hermawan Setiawan, Lytio Enggar Erlangga, and Ido 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)*. IEEE, 2020, pp. 471–475. DOI: 10.1109/ICOIACT50329.2020.9332116.

[Shk19a] Yuri Shkuro. *Embracing context propagation - JaegerTracing - Medium*. Ed. by medium. 2019. URL: https://medium.com/jaegertracing/embracing-context-propagation-7100b9b6029a (visited on 09/21/2021).

[Shk19b] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. 1st ed. Packt Publishing, 2019.

[Spe05] Kevin Spett. *Cross-Site Scripting: Are your web applications vulnerable?* Ed. by SPI Dynamics. 2005. URL: https://people.cis.ksu.edu//~hankley/d764/Topics/SPIcross-sitescripting.pdf (visited on 09/20/2021).

[Sri+11] Manu Sridharan et al. "F4F: Taint Analysis of Framework-based Web Applications". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. Ed. by Cristina Videira Lopes. New York, NY: ACM, 2011, pp. 1053–1067. DOI: 10.1145/2048066.2048145.

[The21a] The Jaeger Authors. *JaegerTracing Documentation: APIs*. 2021. URL: https://www.jaegertracing.io/docs/1.26/apis/ (visited on 09/21/2021).

[The21b] The Jaeger Authors. *JaegerTracing Documentation: Architecture*. 2021. URL: https://www.jaegertracing.io/docs/1.26/architecture/ (visited on 09/21/2021).

[The21c] The Jaeger Authors. *JaegerTracing Documentation: Getting Started*. 2021. URL: https://www.jaegertracing.io/docs/1.26/getting-started/ (visited on 09/21/2021).

[Tri+09] Omer Tripp et al. "TAJ". In: *ACM SIGPLAN Notices* 44.6 (2009), pp. 87–97. DOI: 10.1145/1543135.1542486.

[van+17] Andrew van der Stock et al. *OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks*. 2017. URL: https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010-2017%20(en).pdf (visited on 09/20/2021).

[Vel18] Miguel Velez. *Taint Analysis*. Pittsburgh, 2018. URL: https://www.cs.cmu.edu/~ckaestne/15313/2018/20181023-taint-analysis.pdf.

[Wal20] Debbie Walkowski. *What Is Cross-Site Scripting?* 2020. URL: https://www.f5.com/labs/articles/education/what-is-cross-site-scripting--xss-- (visited on 09/20/2021).

[WD12] Jeff Williams and Arshan Dabirsiaghi. *Interactive Vulnerability Analysis Enhancement Results: December 2012 Final Technical Report*. Ed. by Air Force Research Laboratory Information Directorate. Rome NY, 2012. URL: https://apps.dtic.mil/sti/pdfs/ADA568544.pdf (visited on 08/02/2021).

[ZAP21a] ZAP Dev Team. *Download ZAP*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/download/ (visited on 09/21/2021).

[ZAP21b] ZAP Dev Team. *Getting Started*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/getting-started/ (visited on 09/20/2021).

[ZAP21c] ZAP Dev Team. *ZAP API Documentation: Using Active Scan*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/api/#using-active-scan (visited on 09/21/2021).

[ZAP21d] ZAP Dev Team. *ZAP Documentation - Active Scan*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/desktop/start/features/ascan/ (visited on 09/21/2021).

[ZAP21e] ZAP Dev Team. *ZAP Documentation - Active Scan Rules*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/desktop/addons/active-scan-rules/ (visited on 09/21/2021).

[ZAP21f] ZAP Dev Team. *ZAP Documentation - Alerts*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/desktop/start/features/alerts/ (visited on 09/21/2021).

[ZAP21g] ZAP Dev Team. *ZAP Documentation - Alerts: X - Frame - Options Header*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/alerts/10020/ (visited on 09/21/2021).

[ZAP21h] ZAP Dev Team. *ZAP Documentation - Passive Scan*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/desktop/start/features/pscan/ (visited on 09/21/2021).

[ZAP21i] ZAP Dev Team. *ZAP Documentation - Spider*. Ed. by OWASP Foundation. 2021. URL: https://www.zaproxy.org/docs/desktop/start/features/spider/ (visited on 09/21/2021).

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Bachelor Thesis titled “Finding vulnerabilities by combining taint analysis and dynamic application security testing - Implementation of an IAST prototype” is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 23rd September 2021

Don Benedikt Diephaus

Consent Form

for the use of plagiarism detection software to check my thesis

Last name: Diephaus **First name**: Don Benedikt
Student number: 442808 **Course of study**: Wirtschaftsinformatik
Address: Alsenstraße 16, 48147 Münster
Title of the thesis: "Finding vulnerabilities by combining taint analysis and dynamic application security testing - Implementation of an IAST prototype"

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 23rd September 2021

Don Benedikt Diephaus