AN IMPROVED STATIC ANALYSIS APPROACH FOR DETECTING INPUT VALIDATION VULNERABILITIES IN WEB APPLICATION

ABDALLA WASEF MOHAMMAD MARASHDIH

UNIVERSITI SAINS MALAYSIA

2025

AN IMPROVED STATIC ANALYSIS APPROACH FOR DETECTING INPUT VALIDATION VULNERABILITIES IN WEB APPLICATION

by

ABDALLA WASEF MOHAMMAD MARASHDIH

Thesis submitted in fulfilment of the requirements
for the degree of
Doctor of Philosophy

September 2025

ACKNOWLEDGEMENT

First of all, I would like to thank Allah, the Most Gracious, the Most Merciful, for giving me the strength and patience to overcome many challenges in order to realise my dreams. I am grateful to Allah for providing me with the perseverance and dedication to complete this study.

I cannot express how grateful I am to my supervisor, Dr. Zarul Fitri Zaaba, for his inspiration, assistance, professional guidance, and commitment to facilitating challenges throughout my academic journey. He provided me with valuable advice and deep insights to help me achieve my goal. I cannot thank him enough and consider myself extremely fortunate to have him as a mentor. My sincere gratitude and appreciation go to the cosupervisor, Dr. Khaled Suwais, for his careful remarks, enlightening help, and academic support.

I would like to express my affection to my beloved parents (Dr. Wasef and Hind) and siblings (Dr. Mohammad, Eman, Noor, and Zain). Thank you for giving me overwhelming patience, love, motivation, support, and inspiration that have greatly facilitated the completion of my thesis. I would also like to thank my beloved wife (Maram) for her support and care for me and my children (Alia'a and Ahed); she is a wonderful and ambitious wife, and I wish her success in her academic journey.

I would like to thank everyone at the University of Sains Malaysia for providing a welcoming learning environment. Gratitude is also extended to friends and colleagues in Malaysia for their valuable collaborations, and I wish them all the best. Finally, my gratitude to the Arab Open University for funding me with the Fundamental Research Grant Scheme, and another gratitude to Sean Business International and the Clean Sheet group, who gave me the support and love during my study journey.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Networks
API	Application Programming Interfaces
AUC	Area under the ROC Curve
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
BiRNN	Bidirectional Recurrent Neural Network
BiLSTM	Bidirectional Long Short-Term Memory
BiGRU	Bidirectional Gated Recurrent Unit
CBOW	Continuous Bag-Of-Words
CFG	Control Flow Graph
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CUTE	Concolic Unit Testing Engine
CVE	Common Vulnerabilities and Exposures
DART	Directed Automated Random Testing
DL	Deep Learning
DOM	Document Object Model
DoS	Denial of Service
EXE	Execution Generated Executions
FNN	Feedforward Neural Network
FN	False Negative
FPI	Feasible Paths Identification

FP	False Positive
GRU	Gated Recurrent Unit
HMMs	Hidden Markov Models
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IBM	International Business Machines
IR	Intermediate Representation
JVM	Java Virtual Machine
JpF	Java PathFinder
KNN	k-Nearest Neighbours
LLVM	Low Level Virtual Machine
LSTM	Long Short-Term Memory
LoC	Line of Code
MAC	Message Authentication Code
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSSA	Minimal Static Single Assignment
NLP	Natural Language Processing
OWASP	Open Web Application Security Project
PCA	Principal Component Analysis
PHP	Hypertext Preprocessor
PWM	Path Weight Method
RNN	Recurrent Neural Networks
RS	Recurrent Structures
SARD	Software Assurance Reference Dataset
SGD	Stochastic Gradient Descent

SMT	Satisfiability Modulo Theories
SQLi	Structured Query Language injection
SQL	Structured Query Language
SSA	Static Single Assignment
SSL	Secure Sockets Layer
STP	Simple Theorem Prover
SVM	Support Vector Machine
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
URL	Uniform Resource Locator
VHF	Variable Handling Function
WVS	Web Vulnerability Scanners
XSS	Cross-Site Scripting

LIST OF SYMBOLS

AV	List of all analysed variables
B	Block
CBV	List of the current block variables
CB	Count the total number of conditional blocks
Cond	List represent the generated conditions
DFD	Number of duplicated feasible paths detected
FD	Number of feasible paths detected
GP	List of all generated paths
Instr	List representing the generated instructions
I	Instruction
LO	Count the total number of logical operators
NFP	List of new paths (non-duplicated)
PW	Path Weight
Paths	List representing the generated paths
SGD	List of all super global variables detected
SGV	List of all super global variables in PHP
SNB	Sum of normal blocks "non-conditional"
TFP	Real number of feasible paths (non-duplicated paths)
T	Time

LIST OF APPENDICES

PENDEKATAN ANALISIS STATIK YANG DIPERTINGKAT UNTUK MENGESAN PENGESAHSAHIHAN INPUT KERENTANAN DALAM APLIKASI WEB

ABSTRAK

Aplikasi web masih terjejas dengan kerentanan walaupun pelbagai kerja telah dilakukan dalam penyelidikan keselamatan web sejak berdekad. Pengesahsahihan input kerentanan yang secara khusus berkait dengan kandungan yang dimasukkan oleh pengguna ke dalam dalam borang web menjadi kategori kerentanan yang paling biasa yang dieksploitasi oleh penyerang. Ancaman yang paling biasa kepada serangan pengesahsahihan input ialah XSS AND SQLi yang berkait dengan kelemahan yang wujud dalam kod sumber aplikasi web. Analisis statik dan kaedah ramalan kerentanan merupakan dua teknik yang paling biasa digunakan untuk menganalisis kod sumber dan meminimumkan kerentanan perisian. Kendatipun begitu, analisis statik masih mengalami keputusan positif palsu yang membawa maksud mengesan kerentanan dalam kod sumber secara realitinya adalah selamat. Di samping itu, model ramalan kerentanan terdahulu mengekstrak semua ciri kod sumber yang terdiri dari ciri tidak berkaitan dengan aliran kerentanan dalam program yang mengurangkan prestasi model. Maka, untuk merapatkan jurang, tesis ini mencadangkan kaedah pengesanan XSS dan SQLi. Pertamanya, kaedah analisis statik dicadangkan untuk mengesan laluan yang boleh dilakasanakan dalam kod sumber PHP. Berdasarkan pengetahuan terkini, tidak ada kaedah PHP atau alat yang wujud untuk mengesan laluan yang boleh dilaksanakan dalam kod sumber PHP. Dengan mengenalpasti laluan yang boleh dilaksanakan mengurangkan kadar positif palsu dalam keputusan analisis statik. Kedua, analisis cemar digunakan untuk menjejak sumber kerentanan, pelaksanaan

kerentanan dan sanitasi laluan yang boleh dilaksanakan. Semasa peringkat analisis cemar, setiap sumber kerentanan dikategorikan kepada satu undang-undang konteks kerentanan yang didefinisikan untuk memastikan keberkesanan kaedah sanitasi untuk mengelakkan kerentanan. Ketiga, ciri relevan kerentanan dari laluan tercemar yang dikesan diekstrak. Ia mengekstrak kaedah penapisan yang digunakan untuk membuang ciri yang tidak relevan kepada aliran kerentanan dalam setiap laluan. Ciri-ciri ini digunakan sebagai input kepada pelbagai algoritma pembelajaran mesin untuk mengesan pengesahsahihan input kerentanan. Dibandingkan dengan pelbagai kaedah lain, prestasi kaedah yang dicadangkan adalah lebih tinggi berbanding kaedah terdahulu iaitu ketepatan, ingatan kembali, kejituan dan ukuran-f adalah 98.67%, 98.1%, 79.9%, and 97.97%, masing-masing. Ini juga menekankan kepentingan menentukan laluan yang boleh digunakan, aliran kerentanan dalam laluan program dan mengurangkan ciri-ciri pengkelas untuk belajar konteks semasa iaitu kerentanan yang ada dalam kod sumber yang akan mengurangkan kadar positif palsu dalam keputusan. Kesimpulannya, pendekatan ini menangani batasan kritikal kaedah statik dan ramalan tradisional. Ia menyediakan rangka kerja yang berskala dan kukuh untuk mengesan kelemahan XSS dan SQLi dalam aplikasi web PHP, menyumbang kepada pembangunan sistem perisian yang lebih selamat dan boleh dipercayai, serta membuka jalan bagi penyelidikan masa hadapan dalam analisis kelemahan web merentas platform dan bahasa lain.

AN IMPROVED STATIC ANALYSIS APPROACH FOR DETECTING INPUT VALIDATION VULNERABILITIES IN WEB APPLICATION

ABSTRACT

Web applications continue to suffer from vulnerabilities, despite decades of extensive research in web security. Input validation vulnerabilities, which arise from insufficiently validated user input in web applications, represent a critical attack vector and rank among the most exploited vulnerability types. The most prevalent threats in this context are Cross-Site Scripting (XSS) and SQL Injection (SQLi), which stem from residual weaknesses in the source code of web applications. Two of the most common techniques used to detect such vulnerabilities are static analysis and vulnerability prediction models. However, static analysis often yields high false positive rates, mistakenly flagging a safe code as vulnerable. Conversely, prior prediction models have extracted all features from the source code indiscriminately, including those unrelated to the vulnerability flow, thereby diminishing overall model performance. To address these challenges, this thesis proposes a novel approach for detecting XSS and SQLi vulnerabilities. First, a static analysis technique is introduced to identify feasible execution paths in the PHP source code, an area currently lacking dedicated tools or methods. Identifying feasible paths significantly reduces false positives in static analysis outcomes. Second, taint analysis is employed to trace the sources of vulnerabilities, confirm their execution, and assess the application of appropriate sanitisation along those feasible paths. During this process, each vulnerability source is mapped to a defined contextual rule to validate the effectiveness of the applied sanitisation mechanisms. Third, vulnerability-relevant features are extracted from the identified tainted paths, and a filtering method is applied to

eliminate features irrelevant to the actual vulnerability flow. The refined feature set is then used as input to various machine learning algorithms for vulnerability classification. The proposed approach significantly outperforms existing methods, achieving an accuracy of 98.67%, a recall of 98.1%, a precision of 79.9%, and an F-measure of 97.97%. These results highlight the effectiveness of incorporating feasible path detection, contextual taint tracking, and feature filtering in enhancing the accuracy and reliability of vulnerability detection. By focusing on relevant code paths and context-specific sanitisation, the approach substantially reduces false positives and improves the model's learning of vulnerability patterns. In conclusion, this approach addresses the critical limitations of traditional static and prediction-based methods. It provides a scalable and robust framework for detecting XSS and SQLi vulnerabilities in PHP web applications, contributing to the development of more secure and dependable software systems and setting the stage for future research in web vulnerability analysis across other platforms and languages.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Web applications are now considered one of the standard platforms for representing data and conducting service releases throughout the World Wide Web. The following are contained in these services: social media, financial, education, banking, news sites, and TV channels. Furthermore, making use of a web application is the simplest way to gather information about anything. All a user needs is an Internet connection to access the web applications from anywhere.

Currently, the three essential programming languages for web application development are Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript, in addition to a server-side language such as Hypertext Preprocessor (PHP) (Mishra et al., 2021; Sotnik et al., 2023). PHP supports a significant portion of websites using server-side scripting, with estimations indicating it comprises around 77.5% of all web applications as of 2024 (Pramod et al., 2024; W3Tech, 2024). The extensive use of this technology may be attributed mostly to its simplicity, open-source nature, and straightforward interaction with web technologies (Yin & Lee, 2023; Truszkowski & Pańczyk, 2022). Nonetheless, despite these benefits, PHP's flexible syntax and weakly structured code might result in inconsistent programming practices, thereby compromising the overall security of web applications (Curie et al., 2019). On the other hand, languages such as Java and C# impose stricter rules and a well-organised development environment, which may decrease the probability of unsafe code (Yin & Lee, 2023; Sakharkar, 2023). Comparative analyses indicate that web applications designed in Java have fewer vulnerabilities than those

created in PHP, indicating a critical difference in security performance (Yin & Lee, 2023; Rafamantanantsoa et al., 2021).

The characteristics of PHP facilitate the study of real-world coding practices and the assessment of how security vulnerabilities emerge in practical applications (Curie et al., 2019; Rafamantanantsoa et al., 2021). Due to the wide range of publicly accessible PHP-based applications, researchers have an extensive dataset for the testing and validation of automated detection techniques (Sakharkar, 2023). This advantage makes the PHP language particularly appropriate for research aimed at enhancing secure development strategies, enabling results that directly impact better coding standards and tools to address known vulnerabilities (Truszkowski and Pańczyk, 2022; Talib and Doh, 2021).

The increase in web application usage has made it attractive to both users and hackers. Web applications store much sensitive data, which hackers can steal and gain monetary benefits from. A malevolent user may exploit this vulnerability in an application to inflict harm on the program owner or its users. By exploiting these flaws (availability), attackers can access and alter users' sensitive information (confidentiality) and trustworthy information (integrity).

As shown in Figure 1.2, approximately 74% of the online web applications contain software vulnerabilities, according to a new set of statistics on web security (Veracode, 2023). Over 69% of these applications have at least one OWASP Top 10 vulnerability, and over 56% of those applications have at least one CWE Top 25 vulnerability. Several web vulnerabilities, such as XSS, SQLi, path traversal, directory traversal, and others, have the potential to steal sensitive user data (OWASP, 2021).

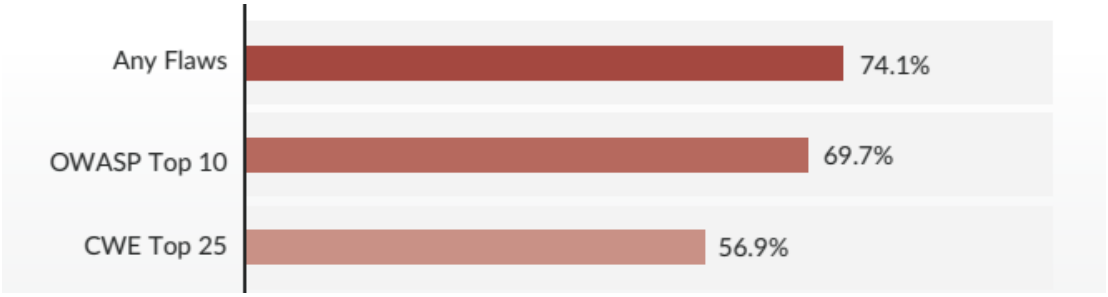

Figure 1.1 Percent of vulnerable applications (Veracode, 2023).

Among all these vulnerabilities, the Open Web Application Security Project (OWASP) includes XSS and SQLi in the list of the 10 most common web application vulnerabilities (OWASP, 2021). SQLi refers to inserting SQL commands or control characters in web inputs to manipulate how SQL queries are executed at the backend. The XSS exploit is implemented by appending script tags to web URLs to trick individuals into clicking malicious links. The exploit is designed to execute on the target system locally. Such attacks exploit the pre-existing user–server trust. Moreover, they benefit from inadequate input/output checks at the server level; if implemented, such checks can halt the execution of malicious JavaScript code. Such vulnerabilities provide malicious techniques, such as SQLi and XSS, to control a web application remotely or cause data theft concerning private user data.

Several solutions have been proposed to detect vulnerabilities in the source code. Despite this, there are still vulnerabilities that exist, and attackers are consistently taking advantage of them (Long et al., 2020; Medeiros et al., 2015; Zhang et al., 2021).

Static code analysis (Louridas, 2006; Reynolds et al., 2017), also known as static program analysis, is the method of examining programs for errors without running them. Developers and testers may use static code analysis to detect vulnerabilities in source code, such as buffer overflows, that are difficult to manually

discover and have the potential to result in security vulnerabilities. On the other hand, static code analysis has been shown to yield a substantial number of false positives (Jovanovic et al., 2006; Technologies, 2021; Yan et al., 2017). False positives examine if the approach incorrectly determines that the code is vulnerable, and false negatives examine if the approach incorrectly determines that the code is invulnerable. The frequency of false positives is high due to (1) code analysis being difficult, and (2) it is frequently preferable for the tool to indicate that there is a problem and be incorrect (i.e., a false positive) than to not state that there is an issue and be incorrect (i.e., a false negative) (Reynolds et al., 2017).

Minimising the occurrence of false positive findings is crucial, as it saves developers time and prevents a decrease in confidence levels in static code analysis tools due to the difficult task of handling a high number of false positives. Developers should prioritise addressing actual warnings and handling them appropriately rather than diverting attention elsewhere (Cheirdari & Karabatis, 2018; Reynolds et al., 2017).

1.2 Motivation

In recent years, there has been an enormous increase in the popularity of web-based applications, which has led to the emergence of various issues within these programs.

The presence of certain issues, such as vulnerabilities, contributes to an increased chance of various attacks that threaten the safety of websites and compromise the integrity of web applications. Consequently, these vulnerabilities present threats to businesses. Security testing is a systematic procedure. The primary objective of this endeavour is to identify security vulnerabilities inside web applications and guarantee the security of relevant web application functionalities. It

is important to acknowledge that web security testing includes the examination of input validation, business logic, authentication, output encoding, and authorisation concerns to mitigate prevalent vulnerabilities, such as SQLi, XSS, and other related threats.

SQLi and XSS are the most common types of web application vulnerabilities (Fedorchenko et al., 2023; Hajar et al., 2024). When an SQLi is initiated, an SQL command is used to send database queries, particularly those that store, retrieve, or delete database data. The attacker manipulates the SQL by targeting it through the comment areas, form input fields, or other user-accessible channels. In other words, the fields that are open for user input allow SQL queries to directly access and query the database. In summary, SQLi is produced as a result of the fields that are accessible for user input, which enable SQL statements to be directly transmitted to and queried by the database. On the other hand, XSS is one of the most frequently occurring vulnerabilities in web applications, with a particular emphasis on the injected scripts in webpages that execute on the server rather than the client.

According to the Group-IB report (Group-IB, 2023), a threat intelligence group, over the period spanning from November to December 2023, an attacker with malicious intent managed to illegally use SQLi to steal over two million email accounts and other forms of personal data from a minimum of 65 identified websites. On the other hand, ESET Research (2024) revealed in October 2023 that the APT organisation Winter Vivern aggressively used XSS vulnerabilities to target European government agencies. Despite the efforts made by researchers in developing approaches and methods to detect these attacks, it is important to acknowledge that XSS and SQLi assaults remain in web applications and continue to pose significant damage to their integrity (Fedorchenko et al., 2023; Hajar et al., 2024).

The persistence of XSS and SQLi causes a danger to international development goals, in addition to its financial and technical implications. For example, SDG 9 (United Nations General Assembly, 2015) illustrates the importance of encouraging innovation and developing infrastructures that are durable. However, insecure digital platforms limit innovation and damage infrastructure. Similarly, SDG 16 (McDermott et al., 2019) argues for accountability, justice, and robust institutions, all of which rely on secure digital platforms to provide services and protect user data. These aims are compromised by vulnerable web applications, which expose essential services to interruption and exploitation.

The vulnerability detection methods use static and dynamic program analysis to identify vulnerabilities in application source code. Static analysis approaches (Jyothi & Lakshmy, 2022; Medeiros et al., 2015; Mohanty & Acharya, 2021; Shashidhara & Madhusudhan, 2021) are able to detect every potential program path in the application by analysing the source code of the web application. However, the primary limitation of static analysis is the high false positive rate in the results, which affects the accuracy of vulnerability detection. A false positive refers to detecting some paths as vulnerable when, in reality, they are safe and contain no vulnerabilities.

Vulnerability prediction models such as Fang et al. (2019), Gupta et al. (2018), and Li et al. (2020) extract source code attributes using static analysis and then apply machine learning to these attributes to detect vulnerabilities. These models, which are both simple and effective, have the capacity to discover vulnerabilities that are missed by more conventional methods of vulnerability identification. They are also used to detect vulnerable code during the code verification phase, which saves time for software testers as they can focus more on the portions of the code designed to protect against vulnerabilities (Einy et al., 2021).

Researchers have urged that code vulnerabilities should be eliminated during the development process in the early stage since attackers may constitute a major danger to the applications if they are exploited at a later stage; it also demands more resources and time to solve the issue. Therefore, enhanced source code analysis has become necessary for finding and addressing these vulnerabilities.

1.3 Problem Statement

Static analysis enables the identification of all possible program paths inside an application via the examination of the source code of the web application. However, the main limitation of static analysis is the associated false positive rate, which may lead to incorrect results and hence impact the final results of the detection of vulnerabilities.

A false positive is defined as results that are safe but are reported as vulnerabilities due to various reasons, such as the detection of infeasible paths as vulnerable paths, while in reality they are infeasible, meaning they will not be executed at all (Aïssat et al., 2016; Jiang et al., 2019). It is a common problem in static analysis approaches. Furthermore, this problem is especially tough in languages like PHP, which are weakly typed and do not have clear specifications for their syntax (Medeiros et al., 2015). The PHP programming language is the primary focus of researchers when it comes to security measures (Fadlalla & Elshoush, 2023). This difficulty is due to its widespread use as a back-end language on the Internet, making it the language of choice for most developers when they begin their programming journey. Based on current knowledge, there is no approach to identifying feasible paths in the PHP source code. In addition, Intermediate Representation (IR) and binary generation tools and libraries are not commonly accessible for all programming languages, such as

Hypertext Preprocessor (PHP), thereby significantly reducing the applicability of current feasible path identification systems (Nguyen et al., 2019).

For detecting vulnerabilities, static taint analysis is the most common information-flow analysis (Grech & Smaragdakis, 2017). Static taint analysis tracks the tainted/untainted status of variables throughout the application control flow. A vulnerability is reported whenever a possibly tainted variable is used in a sensitive (sink) statement without being validated (Avancini & Ceccato, 2010). However, there are various contexts in which the vulnerability might exist in the programs (i.e., inside different HTML contexts), and each of these contexts requires different sanitisation methods to be escaped and sanitised. During analysing the program paths, the knowledge of HTML context is very important to apply the context-sensitive sanitisations (Jaiswal et al., 2014).

On the other hand, the majority of the current analysis tools (de Sousa Medeiros, 2016; Jurásek, 2018; Maskur & Asnar, 2019; Technologies, 2021) and prediction models (Fang et al., 2019; C. Li et al., 2020; Z. Li et al., 2021) do not take the HTML context into account while performing the analysis, which increases the possibility of producing false positive and false negative results (Maurel et al., 2022; Usha et al., 2020). These approaches rely on predefined static rules. They considered the code secure and free of vulnerabilities as long as it used PHP's built-in sanitisation functions (such as *htmlspecialchars* and *htmlentities*) to check user input. An output statement in a dynamic web application takes user input and uses constant HTML strings to create a dynamic response. This combination shows that using the standard built-in function alone is not always enough to prevent vulnerabilities in the context of HTML. In addition, these approaches centre their attention on locating the unsanitised inputs in the application code, but this alone is insufficient to identify all the

vulnerability contexts in the applications (Hauzar & Kofron, 2012; Saxena et al., 2011).

Recently, vulnerability prediction models have emerged as a technique to discover vulnerabilities. Static taint analysis was used in previous studies (Fang et al., 2019; Gupta et al., 2015; Z. Li et al., 2021) to extract all the features from the source code, and then these extracted features were sent as inputs (datasets) to Machine Learning (ML) to detect vulnerability. However, due to their use of static taint analysis, their extracted features contain all the information about the program, including features from the infeasible paths and the features that have not been validated in the vulnerability context during taint analysis. Furthermore, among all the extracted features, several are unrelated to the flow of the vulnerability in the programs (i.e., comments, normal text variables, and functions that are not related to the vulnerability source). Ali et al. (2019) discovered that irrelevant features reduce the precision rate of a classification process, while Spens and Lindgren (2018) mentioned that by minimising the number of irrelevant features considered for classification, the classifier's accuracy is likely to increase.

To summarise, this research pertains to the vulnerability detection approaches for web applications and the factors that affect their results for the following reasons:

Firstly, static analysis still suffers from high false-positive results due to the existence of infeasible paths (the paths that will not be executed at all under any user inputs) in the program. Addressing feasible paths using intermediate representation, binary generation tools, and libraries is not commonly accessible for all programming languages, such as PHP, thereby significantly reducing the applicability of current execution systems when applied to different web technologies (Nguyen et al., 2019).

Secondly, the PHP language is highly dynamic, including built-in variable handling functions, which provide programmers with substantial flexibility. Analysing the variable handling functions will develop the ability to increase the conditions that could be analysed and reduce the false-positive rate in vulnerability detection results. In addition, considering all the user inputs that have been accepted by any of the standard functions (i.e., *htmlspecialchars* and *htmlentities*) as safe variables, some of these functions are unsuitable for escaping some cases of vulnerabilities and thus need to be verified based on specific rules.

Thirdly, due to the use of static taint analysis in extracting features for vulnerability prediction models, the extracted features include all program information, infeasible path features, and elements unrelated to the actual flow of the vulnerability within the program. Extracting all the source code features, including the ones unrelated to the vulnerability, leads to a decrease in the performance of the vulnerability prediction models (Hong et al., 2020; Jin et al., 2005; Zhang et al., 2018).

1.4 Research Questions

There are three research questions based on the research problem statements stated in the previous section:

1. What is the approach to detect the feasible paths in PHP?

2. How to enhance tracking the vulnerabilities in static taint analysis results (tainted/untainted paths) in the detected feasible paths?

3. How to enhance the extraction of features from tainted/untainted paths results with a filtering method?

1.5 Research Objectives

The objective of this thesis is to present an approach that helps construct safe web applications. In particular, the following goals are set to propose an approach to detect the vulnerabilities that will help this thesis reach its main goal:

1. To propose a technique for addressing the feasible paths in PHP source code.
2. To enhance the vulnerability context and sanitisation function among the detected feasible paths.
3. To propose a filtering method that eliminates irrelevant features from tainted paths and reduces the false positive rate in detecting input validation vulnerabilities.

1.6 Research Scope

This research aims to propose an enhanced approach to detecting input validation vulnerabilities in PHP web applications. The proposed approach is limited to detecting XSS and SQLi in PHP web applications, and the evaluation of the proposed approaches was done based on several performance metrics (i.e. accuracy, precision, recall, false positives), as presented in Table 1.1.

Table 1.1 Scope of the research.

No	Item	Scope of Research
1	Web technology	PHP
2	Vulnerability category	Input validation vulnerabilities
3	Vulnerability type	XSS & SQLi
4	Performance Metrics	False positive, precision, recall, Accuracy

1.7 Research Contributions

The main contribution of this research is enhancing the detection of input validation vulnerabilities by proposing an approach that is designed to more accurately identify these vulnerabilities in web applications. The contributions of this research are summarised as follows:

- A new approach to detect the feasible paths in PHP, which significantly increases the applicability of the current feasible path identification system, since there is no approach available to identify the feasible paths in PHP.

- An enhanced taint analysis approach to validate only feasible paths and ensure that the sanitisation functions used are appropriate for sanitising each vulnerability context.

- A new filtering method is proposed to eliminate features unrelated to the vulnerability flow from the extracted features of the proposed static taint approach. As a result, it provides the necessary features for the model to learn the vulnerability flow and help improve detection results.

A systematic mapping view is shown in Figure 1.2 to indicate the mapping among research gaps, connected with their related objectives and contributions.

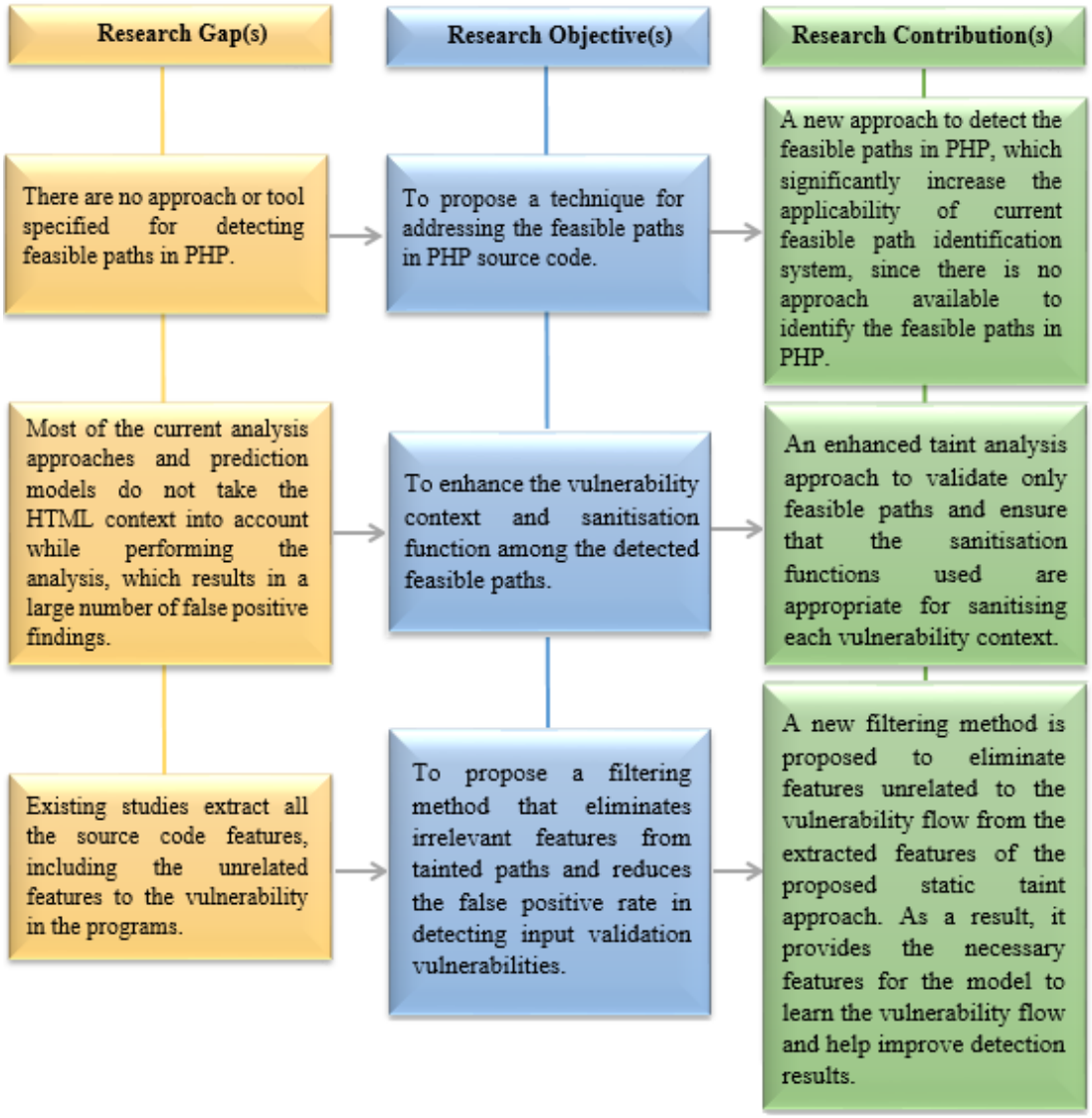

Figure 1.2 Mapping among research gaps, objectives and contributions.

1.8 Research Steps

This thesis proposes an enhanced approach to detecting input validation vulnerabilities in PHP web applications. Figure 1.3 summarises the steps used by this thesis to satisfy the objectives. These steps include the following:

The first stage is the literature review. It reviews the background of input validation vulnerabilities, more specifically, XSS and SQLi. Additionally, it provides an analysis of the existing solutions to detect these vulnerabilities and highlights the

limitations and gaps of these solutions. Then, the research problem is identified based on these limitations and gaps.

The second stage shows the proposed solution, which identifies the proposed solution and its steps in detail to achieve the research objectives. The proposed solution consists of four main steps to detect input validation vulnerabilities with a high detection rate and a low false-positive rate. The proposed solution's main steps consist of data preprocessing, feasible path detection based on symbolic execution, tracking the vulnerability in the detected feasible paths using taint analysis, and finally, extracting the relevant vulnerability features and removing the features that are unrelated to the vulnerability among each detected path. These features will be the target for various ML algorithms to predict the vulnerability in the programs.

The third stage is the design and implementation of the proposed approach, which shows the full details of the design and implementation of the proposed approach. In addition, it provides information about the resource environment, the inputs and outputs of each stage, and the resources used to conduct the implementation of the proposed approach.

The fourth stage is performance evaluation. At this stage, the results of the proposed approach are presented first, followed by an evaluation and comparison with other related solutions based on predefined evaluation metrics.

The fifth stage is the conclusion, where all the contributions and limitations of the proposed approach and future work are presented.

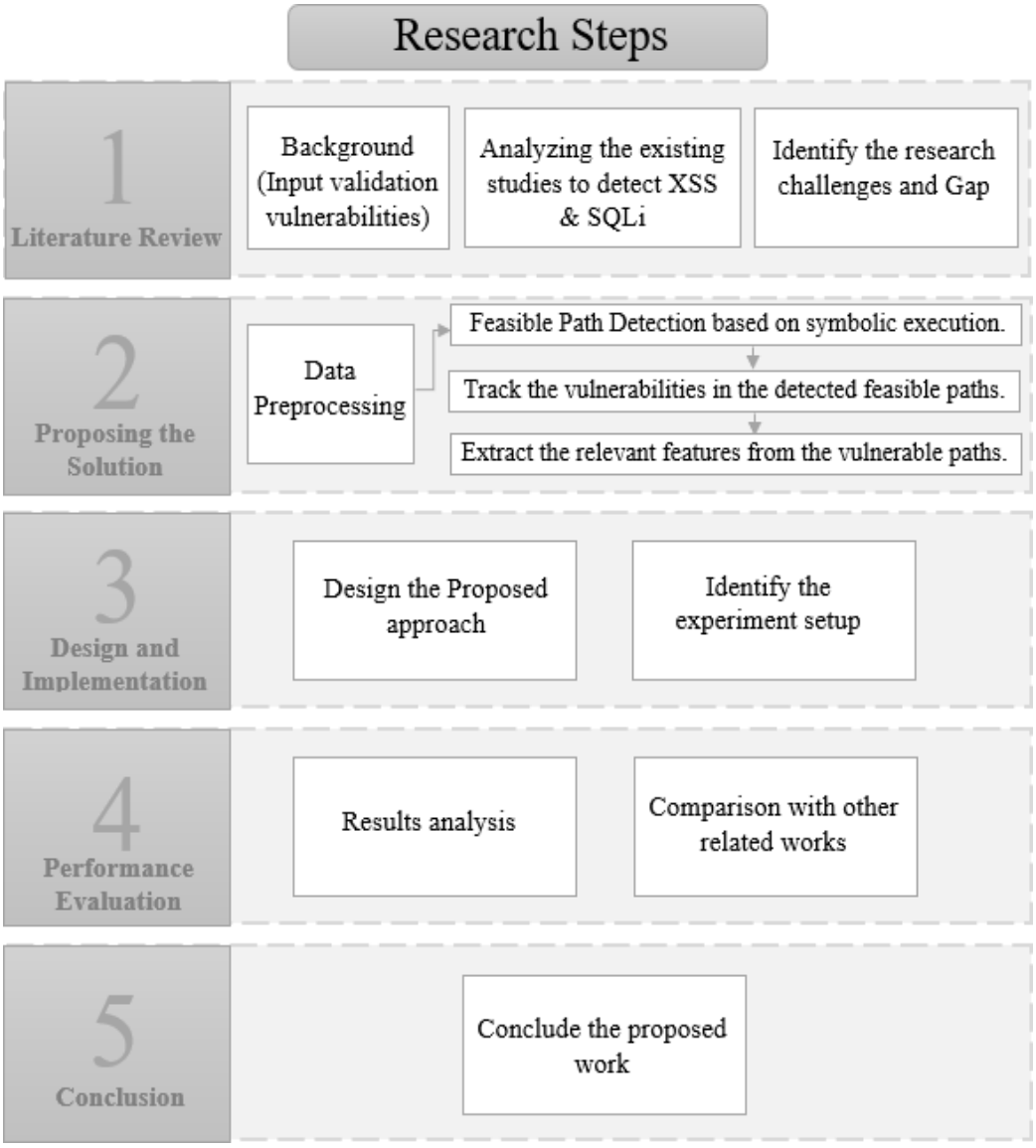

Figure 1.3 Research steps.

1.9 Thesis Organization

The structure of this thesis is as follows:

Chapter 2 includes a comprehensive review of web application security and vulnerabilities. It elaborates on XSS and SQLi vulnerabilities and explores the limits of typical sanitisation procedures in the HTML context, where user input is addressed in output statements. Then, it presents the existing solutions to detect XSS and SQLi

vulnerabilities in web applications. In addition to this, it analyses the benefits and drawbacks of the already available methods.

Chapter 3 describes the proposed methodology, starting with data preprocessing and identifying feasible paths, denoted as Feasible Paths Identification (FPI) in the PHP source code. Followed by a combination of FPI and taint analysis to track the vulnerabilities in the source code, it shows the validation stage of the sanitisation functions within the flow of the program. It describes the proposed feature extraction method, which extracts all features related to the flow of vulnerability among the generated tainted paths and eliminates those that don't make sense.

Chapter 4 covers all the details of the proposed approach, tools, and implementation environment in detail.

Chapter 5 describes the experiments and the results. It also provides a comprehensive evaluation of the results obtained using the proposed approach. In addition, this chapter compares the performance of the proposed approach to that of existing approaches.

Chapter 6 gives the thesis’s conclusion and also includes a discussion of some potential future research.

CHAPTER 2

LITERATURE REVIEW

2.1 Background

A web application is considered a distributed application that one can perform over a web platform. It is a vital component of the current web ecosystem, as it allows for dynamic service and information delivery.

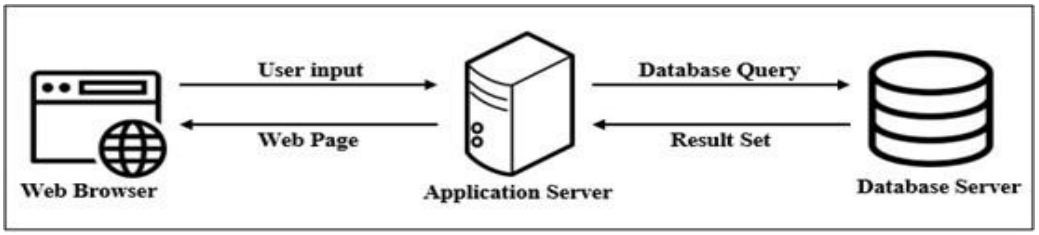

Figure 2.1 System architecture of web applications.

Figure 2.1 demonstrates that this three-tiered architecture is made up of a web browser, which serves as the user interface; a web application server that is responsible for managing the business logic; and a database server for managing the persistent data. Input is received by the web application server in the form of strings taken from two other tiers: input from the user's browser and result sets from the database server. It is common practice to include some of this input in output that is delivered to higher tiers via strings, such as database server queries and HTML pages. Code is constructed dynamically by the web application server. Therefore, the code for the whole web application cannot be found in any one place at any one time so that any one entity can regulate it.

Most web applications have security threats and vulnerabilities that allow attackers to take advantage of them and perform attacks against them. Several studies

have attempted to reduce these attacks by employing various security mechanisms, including encryption devices, intrusion detection systems, and web application firewalls. The next sections outline and elucidate web application security and how to stay safe from such security assaults.

2.1.1 Web Application Security

A secure web application must meet the desired security properties based on a given threat model. For the field of web application security, Li and Xue (2011) consider two threat models as follows: 1) the actual web application is benign (i.e., not owned or hosted for malicious uses) and hosted on a hardened and trusted infrastructure (i.e., the trusted computing base, which includes the web server, operating system, and interpreter); 2) the attacker can manipulate either the sequence or the contents of web requests that the web application receives, but it cannot compromise the application code or the infrastructure directly.

Vulnerability is defined as "a case of an issue in the architecture, deployment, or implementation of software that may violate the security procedures" (Krsul, 1998). Other related concepts, such as bug, fault, and attack, need to be defined and differentiated. A fault is an improper step or procedure in a program that results in unwanted software behaviour (Shin and Williams, 2013). A bug is a fault in a program that leads it to act unexpectedly and serves as proof of a defect (Jalote, 2008). It presents a vulnerability in the automatically triggered software. A vulnerability is a subtype of bug that a hostile person exploits. The vulnerabilities discovered inside the implementation of web applications may undermine the intended security features and make attacks viable.

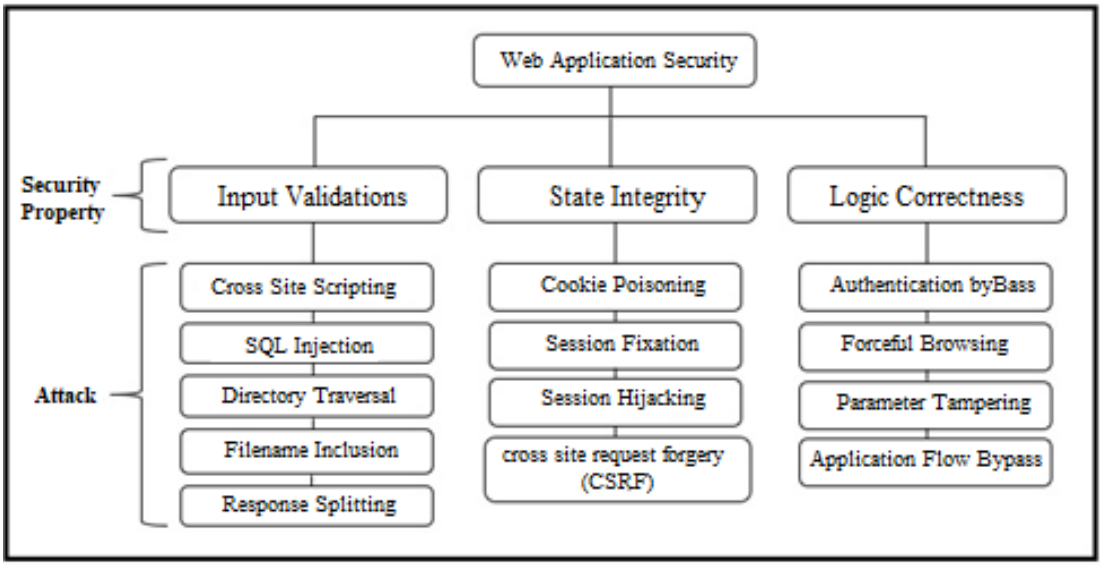

Figure 2.2 Web application security properties

Specifically, a secure web application must be able to maintain the stack of security properties illustrated in Figure 2.2. Input validity signifies that there should be a validation of the user input before it is used by the web application; state integrity means that the state of the application must remain untampered; logic correctness means that there should be proper execution of the application logic based on the intention of the developers (Li and Xue, 2011). The next sections will describe the three security properties and illustrate how the distinct features of web application development can make the security design for web applications more complex.

2.1.1(a) Input Validations

Considering the threat model, one cannot trust the user input data (Li and Xue, 2011). Nevertheless, the application must first validate the untrusted user data (e.g., composing SQL queries or web responses) before utilising it.

The validation of user input is often conducted via sanitisation routines, which turn untrusted user input into trusted data. It does this by filtering suspicious constructs

or characters inside the user input. Although simple in theory, achieving the accuracy and completeness of user input sanitisation is not trivial, especially when scripting languages are used to program the web application. First, because user input data proliferates throughout the application, it needs to be tracked all along so that all the sanitisation points can be identified. However, there has to be proper handling of the dynamic characteristics of scripting languages to ensure that user input data is correctly tracked. Second, the context has to be taken into account during proper sanitisation. This parameter specifies how the user input is used by the application and later interpreted by either the SQL interpreter or the web browser. Therefore, various contexts require unique sanitisation functions. However, the weak typing feature possessed by programming languages makes context-sensitive sanitisation error-prone and challenging (Li and Xue, 2011).

2.1.1(b) State Integrity

State maintenance serves as the basis for constructing stateful web applications, which require a secure web application for maintaining the integrity of application states. However, when an untrusted party (client) is involved in the application state maintenance, ensuring state integrity becomes a challenging dilemma for web applications.

Several attack vectors target the vulnerabilities found in the state maintenance and session management mechanisms of web applications, which include session fixation (predictable session identifier), cookie poisoning (tampering with the cookie information), and session hijacking (stolen session identifier) (Li and Xue, 2011). Another prevalent attack in this category is cross-site request forgery (i.e., session riding). In this kind of attack, the victim is tricked by the attacker into sending crafted

web requests using the valid session identifier of the victim but on behalf of the attacker. This may lead to tampering with the victim's session, disclosing sensitive information (e.g., Salami et al., 2021), and financial losses (e.g., an attacker could forge a web request instructing a vulnerable banking website to move the victim's money to his account).

2.1.1(c) Logic Correctness

Ensuring that logic is correct is vital to the functioning of web applications. Because the application logic is unique to every web application, encompassing all the aspects with just one description is not possible. Instead, it offers a general description that includes the majority of the common application functionalities as follows. It can be challenging to correctly enforce and implement application logic as a result of its mechanism for state maintenance and the "decentralised" framework of web applications.

First of all, the interface-hiding technique, which adheres to the "security by obscurity" principle, is largely deficient in nature. This gives the attacker the ability to reveal hidden links and gain direct access to unauthorised operations or information. Second, developers manually perform the explicit checking of the application state in an ad hoc manner. Thus, there is a high likelihood for certain state checks to be absent on unexpected control flow paths because the web application has many entry points. Furthermore, writing correct state checks can be prone to error since it requires the consideration of both static security policies and dynamic state information. These missing and faulty state checks introduce logic vulnerabilities into the applications (Li and Xue, 2011).

2.1.1(d) Summary

Most web applications have security vulnerabilities that allow attackers to take advantage of them and orchestrate attacks against them. Based on recent reports (OWASP, 2021; Whitehat, 2021) and other associated research (Fang et al., 2019; C. Li et al., 2020; P. Li et al., 2021), the present top vulnerabilities are associated with input validation (i.e., XSS and SQLi). Numerous expensive security breaches to organisations may be a result of the failure to shield web applications from inputs that are invalid (Martin & Lam, 2008). Moreover, it has been stated by Thankachan et al. (2014) and Martin & Lam (2008) that unchecked input validation is a main source of web application attacks. Hacking procedures may lead to the theft of sensitive data, privilege escalation, the destruction of web applications, and unauthorised access to a system. Based on the recent research and security reports (Acunetix, 2022; OWASP, 2021; Ravindran et al., 2023; Whitehat, 2021), XSS and SQLi vulnerabilities are among the top web application vulnerabilities in web applications. Figure 2.3 shows that XSS and SQLi occupied the top two spots recently.

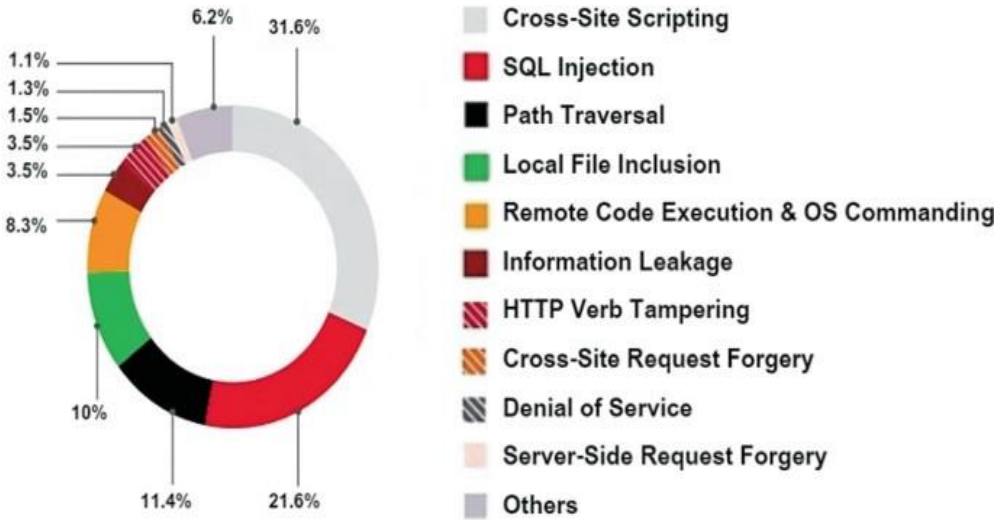

Figure 2.3 List of top web application vulnerabilities (Ravindran et al., 2023)

Web Vulnerability Scanners (WVS) are tools that employ dynamic testing. These scanners are used by software engineers and cybersecurity specialists to identify vulnerabilities in web applications. These scanners can automatically assess the security of web applications with little human effort and are often promoted as 'point-and-click pentesting' solutions. There are several WVSs available, both commercial and free/open source, to assist developers and security researchers in discovering vulnerabilities in web applications. Table 2.1 shows the detection results of these tools in the research articles that were conducted.

The results indicate that SQLi and XSS are the most commonly identified vulnerabilities from the OWASP Top 10 list. It can also be shown that the vulnerability detection rates of the assessed WVSs range from 0% to 100% for detecting SQLi and from 6% to 100% for detecting XSS. XSS and its vulnerabilities are thoroughly discussed and explained in the next section.

Table 2.1 Recent detection rate results per OWASP top ten vulnerability type.

Author	**Web Vulnerability Scanner**	**Vulnerability Type**	**Percentage** %
Aliero et al. (2020)	Acunetix web vulnerability scanner (Acunetix, 2005)	SQLi	80%
	IBM Security AppScan (IBM, 2022)	SQLi	80%
Đurić (2014)	Acunetix web vulnerability scanner (Acunetix, 2005)	SQLi	50%
		XSS	50%
Wagner (2011)	Burp Suite Pro PortSwigger (PortSwigger, 2003)	SQLi	89%
Martirosyan (2012)	Acunetix web vulnerability scanner (Acunetix, 2005)	XSS	72%
		Broke Authentication and Session Management	50%
		Insecure Direct Object Reference	100%
		Insecure Cryptographic Storage	28%
		Insufficient Transport Layer Protection	29%
Garn et al. (2014)	Burp Suite Pro PortSwigger (PortSwigger, 2003)	XSS	89%
Shah (2020)	Burp Suite Pro PortSwigger (PortSwigger, 2003)	SQLi	50%
Suteva et al. (2013)	NetSparker invicti (Invicti, 2009)	SQLi	57%
		XSS	60%

2.1.2 Cross Site Scripting (XSS) Vulnerabilities

XSS is a JavaScript code injection attack that allows an attacker to inject malicious JavaScript into a victim's web browser in order to gain access to sensitive resources such as passwords, cookies, and credit card numbers (Martin & Lam, 2008). To exploit the vulnerabilities of XSS on the web applications, a malicious JavaScript payload is crafted and injected by the attacker on the web application. Injection of this