Mitigating SQL injection and cross site scripting vulnerabilities using program analysis and data mining techniques

Shar, Lwin Khin

2013

Shar, L. K. (2013). Mitigating SQL injection and cross site scripting vulnerabilities using program analysis and data mining techniques. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/55051>

<https://doi.org/10.32657/10356/55051>

**MITIGATING SQL INJECTION AND CROSS SITE
SCRIPTING VULNERABILITIES USING PROGRAM
ANALYSIS AND DATA MINING TECHNIQUES**

SHAR LWIN KHIN

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2013

SHAR L.K.

**MITIGATING SQL INJECTION AND CROSS SITE
SCRIPTING VULNERABILITIES USING PROGRAM
ANALYSIS AND DATA MINING TECHNIQUES**

SHAR LWIN KHIN

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2013

ACKNOWLEDGMENT

The years I spent on my PhD study in NTU have without doubt been the most rewarding part of my life. Campus life, research challenges, working with professors and colleagues have equipped me with necessary skills to thrive. I am eternally grateful to the people who shared these memories with me and to those who made it possible.

My deepest gratitude goes to my supervisor, Prof. Tan Hee Beng Kuan. Tan has provided me with invaluable guidance, continuous support and encouragement. He has spent much effort on finding different research topics and approaches to make progresses and achievements for my project. His insistence on excellence has influenced my way of life and made me a fine researcher, writer, and critical thinker. I also thank Prof. Lionel C. Briand from SnT Centre, University of Luxembourg, for his insightful suggestions and advices. He has shared with me his unique problem solving skills.

My special thanks go to my fellow research students, Ding Sun, Bindu, and Kaiping, who have shared with me some of their research works and with whom I have exchanged ideas.

I would also like to thank my colleagues in InfoComm Research Lab for many fruitful discussions about the research issues and for sharing ideas about the project and contributing suggestions. I am also grateful to the Master students from Communication Software and Networks course who have worked with me in developing prototype tools and carrying out experiments.

Last but not least, I wish to thank my parents and Thinzar, my lovely wife, to whom I am eternally indebted. I am grateful for all their love, care, encouragement, and support throughout this journey. I would not have come this far without them.

TABLE OF CONTENTS

ACKNOWLEDGMENT.....	ii
TABLE OF CONTENTS	iii
SUMMARY.....	vi
LIST OF FIGURES.....	viii
LIST OF TABLES	x
Chapter 1 INTRODUCTION	1
1.1 Motivation.....	1
1.2 Objectives.....	3
1.3 Overview of Our Research.....	3
1.4 Major Contributions.....	5
1.5 Thesis Organization	6
Chapter 2 BACKGROUND	8
2.1 Vulnerability	8
2.1.1 SQL Injection Vulnerability.....	9
2.1.2 XSS Vulnerability	12
2.2 Program Analysis.....	15
2.2.1 Static Program Analysis.....	15
2.2.2 Dynamic Program Analysis	17
2.3 Data Mining	18
2.3.1 Data Preprocessing.....	19
2.3.2 Data Analysis	20
2.3.3 Evaluation	21
Chapter 3 SUPERVISED VULNERABILITY PREDICTION FROM STATIC CODE ATTRIBUTES ...	23
3.1 Research Hypothesis	25
3.2 Static Code Attributes	26
3.2.1 Backward Static Program Slice.....	27
3.2.2 Classification.....	28
3.2.3 Example	32
3.3 Vulnerability Prediction Framework.....	34

3.4	Prototype Tool and Dataset.....	39
3.5	Evaluation	41
3.5.1	Result and Discussion	42
3.5.2	Threats to Validity	47
3.6	Conclusion	48
Chapter 4 SUPERVISED AND UN-SUPERVISED VULNERABILITY PREDICTION FROM HYBRID CODE ATTRIBUTES.....		49
4.1	Hybrid Code Attributes	51
4.2	Research Hypotheses	56
4.3	Vulnerability Prediction Framework.....	57
4.3.1	Data Preprocessing.....	57
4.3.2	Building Supervised Vulnerability Predictors.....	57
4.3.3	Building Un-Supervised Vulnerability Predictors	59
4.4	Example	60
4.5	Prototype Tool and Dataset.....	63
4.6	Evaluation	65
4.6.1	Result on Supervised Vulnerability Prediction	66
4.6.2	Result on Un-Supervised Vulnerability Prediction.....	68
4.6.3	Threats to Validity	69
4.7	Conclusion	70
Chapter 5 VULNERABILITY AUDITING.....		71
5.1	Code-based Extraction of SQLI and XSS Defense Artifacts.....	73
5.1.1	Characterizing Defense through Validation.....	75
5.1.2	Characterizing Defense through Escaping	76
5.1.3	Characterizing Defense through Filtering	77
5.1.4	Characterizing Defense through Character Set Specifying.....	77
5.1.5	Security Checks and Operations	78
5.2	Auditing the Defense against XSS and SQLI Threats	79
5.2.1	Extraction of Defense Features	79
5.2.2	Auditing the Defense.....	82
5.3	An Example Illustrating the Proposed Approach.....	84
5.4	Evaluation	88
5.4.1	Prototype Tool.....	89

5.4.2	Experiment	91
5.4.3	Result	92
5.4.4	Limitation	95
5.5	Conclusion	96
Chapter 6 XSS VULNERABILITY REMOVAL		97
6.1	XSS Prevention Rules	98
6.2	Proposed Approach	101
6.2.1	Detection	101
6.2.2	Removal	103
6.3	Evaluation	108
6.3.1	Prototype Tool	109
6.3.2	Test Subjects	110
6.3.3	Experiment	111
6.3.4	Result and Discussion	114
6.3.5	Limitation	116
6.4	Conclusion	117
Chapter 7 RELATED WORK		118
7.1	Defensive Coding Practices	119
7.2	Data Mining Techniques	121
7.3	Static Analysis-based Vulnerability Detection	125
7.4	Vulnerability Testing	128
7.5	Runtime Attack Prevention	131
7.6	Summary	134
Chapter 8 CONCLUSIONS AND RECOMMENDATIONS		136
8.1	Contributions towards Academic Research	136
8.2	Contributions towards Industry Practice	139
8.3	Recommendations	141
APPENDIX A		145
AUTHOR'S PUBLICATIONS		146
BIBLIOGRAPHY		147

SUMMARY

This thesis presents approaches for mitigating SQL injection (SQLI) and cross site scripting (XSS) vulnerabilities, the two most common vulnerabilities found in web applications in recent years.

Current approaches to mitigate SQLI and XSS problems can be broadly classified into three types which are defensive coding, vulnerability detection, and attack prevention. Defensive coding approaches provide input validation and input sanitization methods that are effective against SQLI and XSS. Vulnerability detection approaches typically focus on identification of vulnerabilities in program source code. Attack prevention approaches focus on warding off real time attacks during runtime.

Although all these approaches are certainly useful and could address some of SQLI and XSS issues, there are some major drawbacks that lead to the continual occurrences of the two vulnerabilities nowadays. Defensive coding approaches are very effective when practiced correctly. However, as these approaches generally require intensive manual work, they are error-prone. Vulnerability detection techniques are useful as they could automatically report vulnerable statements in programs. But these techniques are known to report many false positive cases while they could also miss some vulnerabilities. As these techniques also lack focus on reporting the defense features implemented in the programs, it is difficult for one to identify those false positive and false negative cases. Attack prevention techniques are an effective and efficient solution for deployed applications. But these techniques do not address vulnerabilities in programs. As more sophisticated attack vectors are being discovered, programs with vulnerabilities not removed always risk the possibility of being exploited anytime.

Hence, it is clear that complementary or alternative solutions, which are easy to be used and yet effective, are required to comprehensively address the threats of SQLI and XSS. Based on these motivations, in this thesis, we propose three complementing novel approaches which are vulnerability prediction, vulnerability auditing, and vulnerability removal.

Overall, all the three proposed approaches are based on program analysis techniques augmented with the use of pattern-based empirical models. In addition, vulnerability prediction approach also involves data mining techniques. We empirically discover interesting patterns from program source code and program execution traces and reflect those patterns in appropriate

models so that the models can be used for vulnerability prediction, auditing, and removal purposes.

Firstly, we propose a vulnerability prediction approach based on static and dynamic program analysis techniques, and data mining-based classification and clustering techniques. We show that a set of code attributes that characterize input validation and input sanitization code patterns in web applications can be used to build useful vulnerability predictors. The attributes are collected using both static analysis and dynamic analysis techniques. Static attributes are easier to collect using conventional static control flow and data flow analysis. But, dynamic attributes can provide more accuracy as they could reflect program behaviors more precisely. Our vulnerability predictors provide an alternative solution to existing vulnerability detection approaches.

As existing approaches do not provide comprehensive information on the implementations of defense artifacts in programs, one has to manually inspect a large chunk of code to identify the deficiencies in current defense implementations in an attempt to fix vulnerabilities. Hence, secondly, to complement existing approaches, we propose a vulnerability auditing approach that systematically recovers SQLI and XSS defense features implemented in program source code for assisting to verification and fixing of SQLI and XSS vulnerabilities. This approach only relies on conventional static program analysis to extract defense features.

Finally, we present a vulnerability removal approach that automates the application of defensive coding methods in program source code. Pattern analysis and data flow analysis techniques are mainly used to apply required defensive coding schemes in appropriate code locations in an automated way.

This thesis also presents experimental evaluations of the proposed approaches and demonstrates that the approaches are useful and effective.

LIST OF FIGURES

Figure 2-1. Sample server-side program and client-side script vulnerable to XSS.	13
Figure 2-2. Example URLs that direct web users to travelingForum..	14
Figure 2-3. A typical confusion matrix	22
Figure 3-1. (a) Sample Web application program that contains vulnerable and non vulnerable sinks (b) Control flow graph representing the backward program slice of statement 21.....	33
Figure 3-2. Architecture of the prototype tool, <i>PhpMiner</i>	40
Figure 3-3. Classification results of (a) XSS vulnerability predictors (b) SQLI vulnerability predictors, built from static code attributes.....	43
Figure 3-4. Best attributes selected by <i>Weka</i> 's gain ratio ranker for the 11 datasets shown in Table 3-3.	45
Figure 3-5. Balance results over 11 datasets of each classifier learned on best 3, 5, 8, or all attributes.....	46
Figure 4-1. (a) Sample vulnerable PHP code (b) Data dependence graph of sink statement.....	61
Figure 4-2. Architecture of enhanced <i>PhpMiner</i>	63
Figure 4-3. Classification results of XSS and SQLI vulnerability predictors built from hybrid code attributes.....	67
Figure 4-4. Classification results of XSS and SQLI vulnerability predictors built from static code attributes.....	67
Figure 4-5. <i>k</i> -means cluster analysis results on the datasets (a) which meet the assumptions (b) which violate the assumptions	69
Figure 5-1. Algorithm for extracting SQLI and XSS defense features from code.....	80
Figure 5-2. (a) Java Servlet code snippet of a Log-in authentication program. (b) The CFG of the program. (c) Tainted-information flow graphs.	85
Figure 5-3. Architecture of the prototype tool, <i>WAVDE</i>	90
Figure 6-1. (a) Code snippet of a sample Login Servlet program. (b) CFG of the program.....	103
Figure 6-2. Sequences of Nodes that are analyzed and HTML document structures collected from string objects	106

Figure 6-3. Login Servlet program secured with ESAPI's security APIs and the report produced by the algorithm.	108
Figure 6-4. Architecture of the prototype tool, <i>saferXSS</i>	110
Figure 7-1. Overview of an ideal use of SQLI and XSS mitigation techniques during a software development process	119

LIST OF TABLES

Table 3-1. Static code attributes	29
Table 3-2. Statistics of the test subjects used	41
Table 3-3. Datasets.....	41
Table 4-1. Static-dynamic hybrid attributes.....	53
Table 4-2. Statistics of the test subjects	65
Table 4-3. Datasets.....	65
Table 5-1. Statistics of the test subjects	89
Table 5-2. Results on SQLI vulnerability auditing	93
Table 5-3. Results on XSS vulnerability auditing.....	93
Table 6-1. XSS Vulnerabilities removed by <i>saferXSS</i> tool.....	112
Table 6-2. Actual XSS vulnerabilities found in the test subjects and potential XSS vulnerabilities reported by <i>saferXSS</i> tool, which correspond to each HTML context type	112
Table 6-3. Example of XSS attacks performed on original and modified test subjects.....	114
Table 6-4. Results of functional and XSS test cases executed on original and modified test subjects	116
Table A-1. Notations used	145

Chapter 1

INTRODUCTION

1.1 Motivation

Today, multi-user web applications are becoming the dominant way to handle many of our daily activities. Thus, ensuring software security is one of the most important industrial practices nowadays because security risks imposed by web application vulnerabilities could have serious and severe consequences. Web application vulnerabilities such as cross site scripting (XSS), SQL injection (SQLI), buffer overflow, path traversal, and open redirect/forward, are commonly found in applications that fail to properly handle user inputs [25, 28, 36, 111]. Among them, for the past decade, SQLI and XSS vulnerabilities have been ranked as the top two common and serious security threats found in web applications [29, 108]. According to security reports [1, 19, 27, 103, 151], these two types of vulnerabilities are even found in widely-used, giant applications such as Google, Facebook, MySpace, Vodafone, Microsoft .NET framework, and McAfee anti-virus.

In literature, a few solutions to mitigate these two serious security threats have been proposed. These solutions can be classified into defensive coding, vulnerability prediction, vulnerability detection, vulnerability testing, and runtime attack prevention techniques.

Defensive coding practice is the best solution since both SQLI and XSS vulnerabilities are the direct consequence of insecure coding methods practiced by developers. Current defensive coding practices [26, 88, 117, 167] provide security libraries and secure software development frameworks, but being manual, these practices are *labor-intensive* and error-prone. And it is also hard to enforce these practices into deployed web applications.

Vulnerability prediction, detection, and testing approaches can detect vulnerabilities and assist security auditors in fixing the vulnerabilities. Vulnerability prediction approaches predict vulnerabilities by applying data mining techniques on software code attributes such as code complexity attributes [87]. A major disadvantage of existing vulnerability prediction approaches [42, 101, 138, 156] is that they are *coarse-grained*, that is, they only report vulnerabilities at

software module or program levels. Therefore, much manual inspection work is still necessary to locate the vulnerable code in reported modules or programs.

Vulnerability detection approaches such as [61, 82, 165] typically apply static program analysis techniques to detect the use of insecure information in security-sensitive program operations. These approaches trace the flow of external user inputs along program paths and through language built-in and user-defined functions. And they generate security alerts if the traces contain security-sensitive operations. Unlike the above vulnerability prediction approaches, these approaches are fine-grained as they can report the vulnerable code traces. But they *lack accuracy* because, being static-based, they are unable to precisely analyze the correctness of the input validation and sanitization functions [9, 68] and thus, they may fail to distinguish between secure data and insecure data.

Vulnerability testing approaches generate test cases that could expose SQLI and XSS vulnerabilities. These approaches are dynamic analysis techniques, and can be categorized into specification-based and code-based. Specification-based approaches [63] generate test cases from known vulnerability patterns. These approaches are useful for quick detection of the presence of vulnerabilities. But they are *not effective* as they could miss vulnerabilities having unspecified or unknown code patterns. Code-based approaches are generally more effective as they generate test cases based on code-based information. But, earlier existing approaches such as [38, 126, 127, 139] are labor-intensive as test case generation and vulnerability inspection are manual. In recent code-based testing approaches such as [9, 68, 73, 85, 120, 160], these processes are automated using string constraint solving, model checking, and concolic execution techniques. However, these approaches are generally *not scalable* because model checking and concolic execution suffer from state space explosion. Solving string constraints can be also very complex and thus, full implementations of some of these approaches are also not available.

Runtime attack prevention approaches [13, 14, 48, 49, 59, 83, 112, 144, 166] focus on preventing real-time attacks using intrusion detection systems. In general, these approaches set up a proxy between the client and server or between the database and server, to intercept the data traffic and verify the payload against security policies. These approaches are intended for deployed applications and are not suitable for locating and fixing the vulnerabilities. They also incur runtime overheads.

Furthermore, all these existing approaches generally lack focus on systematic reporting of the defense artifacts implemented in a program. Therefore, even though existing techniques may help detect vulnerable program statements, it is still a tough task for security auditors to debug or fix

the reported vulnerabilities. They may still have to examine the adequacy of the defense features implemented in web programs through inspecting a large chunk of program source code.

Hence, we can summarize that existing SQLI and XSS mitigation approaches suffer from one or more of the following weaknesses—(1) intensive labor, (2) coarse-grained analysis, (3) inaccuracy, (4) in-scalability, (5) runtime overhead, (6) lack of program verification assistance, (7) incomplete implementation. As a result, software development teams may face difficulties in adopting these existing approaches.

1.2 Objectives

The overall research objective of this thesis is to explore the application of empirical knowledge, static analysis, dynamic analysis, and data mining techniques in addressing the software security issues caused by SQLI and XSS vulnerabilities. Motivated by the observations highlighted in the above section, we aim to mitigate the weaknesses of existing approaches and leverage their strengths. Thus, we shall propose novel approaches that augment and improve existing vulnerability mitigation approaches, and also develop tools that implement the proposed approaches so that the proposed approaches can be applied to real-world applications and can help software development professionals in ensuring software security.

1.3 Overview of Our Research

Program analysis plays an important role in most software engineering activities such as testing, debugging, reverse engineering, and program comprehension. It can be used to extract properties or characteristics of functions and statements in a program. In recent years, data mining techniques have also been successfully used in software engineering tasks such as predicting software quality [10]. On the other hand, the use of program analysis augmented with empirical studies has been successful in solving problems of software testing and maintenance issues [79, 102, 121]. They showed that important properties of software systems can be inferred through the use of empirical knowledge in program analysis more effectively than the use of traditional analyses.

Hence, in this research, we explore the use of empirical knowledge, program analysis, and data mining techniques in mitigating SQLI and XSS vulnerabilities. From our surveys on SQLI and XSS mitigating approaches [130, 135], we observed that every individual type of vulnerability mitigation approaches has its own shortcoming. For example, as we discussed above, defensive coding practices are effective but, these practices are prone to human errors. Hence, we noted that

SQLI and XSS threats have to be defended from a combination of different types of vulnerability mitigation approaches. As a result, in this thesis, we propose three different types of approaches—*vulnerability prediction*, *vulnerability auditing*, and *vulnerability removal*. Each approach provides an important step in nullifying the threats of SQLI and XSS vulnerabilities.

Overall, all the three proposed approaches are based on program analysis techniques augmented with the use of pattern-based empirical models. In addition, vulnerability prediction approach also involves data mining techniques. We empirically discover *interesting patterns* from program source code and program execution traces and reflect those patterns in appropriate models so that the models can be used for vulnerability prediction, auditing, and removal purposes. In our context, we define that a code pattern or a program execution behavior is interesting if it can be useful for mitigating SQLI and XSS vulnerabilities.

Firstly, we propose a novel vulnerability prediction approach. We perform empirical studies on many web applications to identify common code patterns and program execution behaviors that result in SQLI and XSS vulnerabilities (vulnerable patterns) and that do not result in those vulnerabilities (non-vulnerable patterns). We model those patterns into a set of attributes so that we can extract those attributes from a given web program using program analyses and then build vulnerability predictors through applying data mining techniques on those attributes.

We show later in the thesis that our vulnerability predictors are effective. But, vulnerability prediction is only a first step of our research because it does not provide information on the deficiencies of security-defense artifacts that cause vulnerabilities. To fix vulnerabilities, this information is essential. Hence, to complement the prediction approach, we naturally propose a novel vulnerability auditing approach that assists security auditors in fixing the vulnerabilities. Again, we use program analysis augmented with empirical models. From empirical studies on a handful of real-world web applications, we identify properties that can be used to recover SQLI and XSS defense models from web programs. We then provide required program analysis techniques to extract those models automatically. We also provide guidelines to perform security audits based on the recovered defense models. Hence, security audits only need to be performed on the recovered models instead of the whole chunk of code, saving time and effort.

Although our vulnerability auditing approach would be very useful to auditors, auditing still requires manual effort and some vulnerabilities would still remain if the auditors misunderstand the reports generated by our auditing approach. Therefore, as a final step of our research, we propose an automated XSS vulnerability removal approach. In literature, Thomas et al. [147] has proposed an SQLI vulnerability removal approach that generates and inserts prepared SQL

statements in web programs. Therefore, although our vulnerability prediction and auditing approaches target both SQLI and XSS, our vulnerability removal approach focuses only on XSS. Similar to our vulnerability auditing approach, we use program analysis augmented with empirical knowledge. We empirically study various code patterns that cause XSS vulnerabilities and categorize them into different types that require different types of security features to prevent vulnerabilities. We then apply static program analysis to detect those code patterns and insert appropriate security features into vulnerable code locations.

Therefore, our vulnerability removal approach complements the above two vulnerability prediction and auditing approaches. More importantly, our research work, as a whole, provides an improved or complementary solution to existing research works in addressing the issues of SQLI and XSS.

1.4 Major Contributions

In this thesis, we propose three main novel approaches that mitigate SQLI and XSS. These approaches have been published in international journals and conferences including the top software engineering conferences such as *ICSE*. The major contributions of this thesis are:

- 1) **Vulnerability prediction:** We propose a novel set of static analysis-based and dynamic analysis-based attributes that characterize common input validation and sanitization code patterns found in web applications. We hypothesize that these attributes are important indicators of SQLI and XSS vulnerabilities.

We build two types of vulnerability prediction models—classification-based and clustering-based, from the proposed attributes. As classification is a type of supervised learning, classification-based prediction models are dependent on the availability of training data labeled with known vulnerabilities. But, as cluster analysis is a type of unsupervised learning, clustering-based models work in the absence of labeled training data. Hence, our proposed vulnerability predictors can be used to predict vulnerabilities both in the presence and the absence of labeled training data.

We prove our hypothesis based on a series of experiments using both types of prediction models. This work has been published in [132, 134, 136].

- 2) **Vulnerability auditing:** We propose a code auditing approach that recovers defense features implemented in program source code into a model, using static program analysis. We also provide guidelines to carry out security audits on the recovered defense models.

We hypothesize that the proposed approach is effective in recovering the defense features implemented in web programs, and is useful in filtering the false alarms reported by existing vulnerability detection methods and fixing the real vulnerabilities.

We prove our hypothesis by conducting SQLI and XSS vulnerability auditing experiments with the help of Master students. This work has been published in [62, 128, 133].

- 3) **Vulnerability removal:** We present an automated vulnerability removal approach which statically removes XSS vulnerabilities from program source code. We first identify potential vulnerabilities in program source code using static program analysis. We then secure those vulnerabilities by inserting adequate defensive coding methods into appropriate code locations. Our hypothesis is that this approach can remove all vulnerabilities identified as potentially vulnerable.

We apply this approach to a set of vulnerable applications and prove our hypothesis. A preliminary version of this work which addresses a different type of web application vulnerability has been published in [129]. The work presented in this thesis has been published in [131].

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the background required. It first reviews SQLI and XSS vulnerabilities. Then, it discusses static and dynamic program analysis techniques and data mining-based learning schemes, which shall be used in our proposed approaches.

Chapter 3 presents our work on vulnerability prediction. We propose a set of *static code attributes* that characterize input validation and input sanitization code patterns. We then build SQLI and XSS vulnerability predictors from the proposed attributes. We also present our prototype tool called *PhpMiner* that collects data from PHP programs. Using the tool, we conduct experiments on a set of PHP applications to evaluate the proposed predictors.

Chapter 4 enhances our static analysis-based vulnerability approach presented in Chapter 3. We propose the use of *dynamic attributes* in addition to static attributes so as to improve accuracy. We discuss methods to collect dynamic attributes from function execution traces. Furthermore, to avoid reliance on the availability of labeled training data to build vulnerability predictors, we present prediction models that are based on both classification and clustering in order to predict vulnerabilities, in the presence or absence of labeled training data, respectively. We then conduct experiments on a set of PHP applications to evaluate our proposed approaches.

Chapter 5 presents the techniques for the security auditing of web application programs. We propose an approach for extracting defensive code features implemented in web programs. We then provide techniques and guidelines for verifying the adequacy of extracted defense features in preventing XSS and SQLI attacks. Finally, we describe our prototype tool called *WAVDE* that automates the extraction of defense features and then, we conduct experiments with the aid of the tool to demonstrate the usefulness of our proposed code auditing approach.

Chapter 6 presents an automated approach that statically removes XSS vulnerabilities from program source code. We provide concepts and techniques for identifying potential vulnerabilities in program source code and determining the required security mechanism to fix the identified vulnerabilities. We also present our prototype tool called *saferXSS* that automates the proposed approach and then, we conduct experiments using the tool to show that the proposed technique is effective in removing XSS vulnerabilities.

Chapter 7 compares the works presented in this thesis with related work. We discuss defensive coding techniques that are effective against SQLI and XSS vulnerabilities, software defect and vulnerability prediction techniques that could identify buggy and problematic code regions with high probabilities, vulnerability detection techniques that identify and locate vulnerabilities in programs, and attack prevention techniques that ward off security attacks at runtime.

Finally, Chapter 8 summarizes the contributions of this research and identifies potential directions for future work.

All the notation conventions used throughout this thesis is listed in Appendix A.

Chapter 2

BACKGROUND

This chapter provides background on SQLI and XSS vulnerabilities, and program analysis and data mining techniques. Section 2.1 reviews SQLI and XSS vulnerabilities, and the injection methods that are typically used to exploit these vulnerabilities. Section 2.2 and Section 2.3 provide an overview of fundamental program analysis techniques and data mining techniques that we shall apply in our proposed approaches to mitigate SQLI and XSS vulnerabilities.

2.1 Vulnerability

Before we discuss the two vulnerabilities, we shall first introduce the definitions and the terms, which shall be used throughout the thesis.

SQL is the standard language for accessing most database servers such as MySQL, Oracle, MSSQL, etc. [6]. *SQL statement* is a statement in a program that uses SQL to create, access, or modify data in a database. *HTML output statement* is a statement in a program that produces an output, which becomes (part of) a client-side document when interpreted by web browser.

User input is an input submitted by an external user into a web application program. As user input is the source of SQLI and XSS attacks, we may address user input also as *tainted data*.

Both SQLI and XSS are a class of *code injection vulnerabilities*, which can be exploited by injecting special characters or keywords that are significant to a script interpreter into program outputs such that the injection causes the program to perform un-intended actions. In SQLI, malicious SQL scripts containing SQL special characters or keywords (such as `'` and `OR`) are inserted into SQL statements via unrestricted user input parameters in order to change the logic of the intended query [6]. Similarly in XSS, malicious client-side scripts (e.g., JavaScript) containing client-side interpreters' special characters or keywords (such as `<script>`) are injected into HTML output statements such that the injected code when executed by browsers could perform malicious actions to the client [109].

In this thesis, we shall use the terms XSS vulnerability and XSSV interchangeably depending on the context. Similarly, we shall also use the terms SQLI vulnerability and SQLIV interchangeably. In the following sub-sections, the two types of vulnerabilities are discussed in details.

2.1.1 SQL Injection Vulnerability¹

SQL threats exist in any web application that accesses database via SQL statements constructed with external input data because an attacker could modify these statements by manipulating the data. Therefore, SQLI is caused by inadequate validation and sanitization of user inputs. An SQLI attack could compromise the database and issue arbitrary SQL commands.

Most web programming languages such as Java, ASP, .NET, and PHP provide a number of methods to construct and execute SQL statements. Due to circumstances such as lack of development time and training, and lack of experience and knowledge of potential security issues, developers often misuse these methods resulting in SQLI vulnerabilities.

Depending on the source of user inputs referenced in SQL statements, SQLI can be classified into two types: first order SQLI and second order SQLI. *First order SQLI* occurs when a web program accesses user input from incoming HTTP request parameters and immediately used it in the SQL code within the same program. *Second order SQLI* occurs when a web program stores an unrestricted user input in a persistent data store such as session objects or database itself and later, a program re-uses the stored data in an SQL code [6].

With some practical examples, the following discusses the insecure programming practices commonly adopted by developers and the injection methods that may be used by attackers to exploit developers' mistakes.

Dynamic query building with string concatenation is commonly used by developers to construct SQL statements. Queries are formed with inputs directly received from external sources during runtime. This method is useful to developers as different queries can be built according to different conditions set by users. However, it is also the cause of many SQLI issues, which is explained in the followings with PHP code examples (`name` and `pwd` are the "varchar" type columns and `id` is the "integer" type column of `user` database table).

¹ This section is reprinted, with permission, from L.K. Shar and H.B.K. Tan, "Defeating SQL injection," IEEE Computer © 2013 IEEE

Absence of check: Developers often use inputs in dynamic query statements without any checks. This is the most common and serious programming mistake. For example, the following PHP code represents such a dynamic SQL statement:

```
$query = "SELECT info FROM user
        WHERE name = '$_GET["name"]' AND pwd = '$_GET["pwd"]'";
```

Attackers could use tautologies to exploit this insecure practice. By supplying the value `x' OR '1'='1` (known as tautology-based attack) to the input parameter `name`, an attacker could access user information without having a valid account because the WHERE-clause condition becomes:

```
WHERE name = 'x' OR '1'='1' AND ...";
```

which is evaluated to be true regardless of the validity of the user name and password provided.

Insufficient escaping: If special characters meaningful to a SQL parser are escaped, they shall not be interpreted as SQL commands. For example, the above tautology-based attack could be prevented by escaping the single-quote `'` character (to avoid being interpreted as a string delimiter) from the inputs used. However, many developers are either not aware of the full list of characters that have special meanings to the SQL parser or not familiar with the proper usage patterns. For example, in the following PHP code, `mysql_real_escape_string` (a MySQL-provided-escaping function) is used to escape MySQL special characters.

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE pwd LIKE '%$pwd%'";
```

The function `mysql_real_escape_string` would protect SQL statements which do not use pattern matching database operators such as `LIKE`, `GRANT`, and `REVOKE`. But, in this case, attackers could include additional wildcard characters `%` and `_` in the password field to match more password characters than the beginning and end characters because `mysql_real_escape_string` does not escape wildcard characters.

Absence of data type check: Another programming mistake made by developers is that data types are rarely checked before SQL statements are constructed. Instead, they tend to apply programming language or database provided sanitization functions such as `addslashes` and `mysql_real_escape_string` that escape known malicious characters such as single-quote `'`. But when the query is to access the database columns of non-text-based data types such as numeric, the attack needs not consist of known malicious characters. For example, the following PHP code shows a SQL statement for which a tautology-based attack could be conducted by supplying the value `1 OR 1=1` to the parameter `id`.

```
$id = mysql_real_escape_string($_GET["id"]);
$query = "SELECT info FROM user WHERE id = $id";
```

For such queries, instead of escaping, data type check (e.g., `if(is_numeric($id))`) should be used to prevent SQLI attacks.

Absence or misuse of delimiters in query string: When a query string is constructed with inputs, proper delimiters have to be used to indicate the data type of the input used. If the delimiters are missed or misused, SQLI could be carried out even in the presence of thorough input validation, escaping, and type checking. For example, in the following PHP code, string delimiters are not used to indicate the input string used in the SQL statement.

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE name = $name";
```

In this case, when the database server has the automatic type conversion function enabled, SQLI attacks could be created using alternate encoding method that would circumvent input sanitization routines. For instance, if an attacker supplies an encoded HEX string `0x270x780x270x200x4f0x520x200x310x3d0x31` to the parameter `name`, the database parser may convert it to the “varchar” value resulting in the tautology string `'x' OR 1=1`. Because the conversion happens in the database, the escaping function used in the server program would not detect any special characters encoded in the HEX string.

Hence, since dynamic query building is often problematic, some developers opt to use parameterized queries or stored procedures which are more secure methods. But, inappropriate use of these methods may still result in vulnerable code, which is discussed in the following.

Improper construction of parameterized queries or stored procedures: Most developers believe that SQLI is impossible when parameterized queries or stored procedures are used to run SQL statements. Although this is true in general, some developers are not aware that SQLI is still possible if parameterized query strings or stored procedures accept non-parameterized inputs. For example, in the following PHP code, although SQLI cannot be conducted through the parameter `name`, it is possible through the parameter `order` which is not parameterized. An attacker may inject piggy-backed query attacks (malicious queries attached to the original query) such as `ASC; DROP TABLE user; --` into the parameter `order`.

```
$query="SELECT info FROM user WHERE name = ?"."ORDER BY '".$_GET["order"]."'";
$stmt = $dbo->prepare($query);
$stmt->bindParam(1, $_GET["name"]);
$stmt->execute();
```

2.1.2 XSS Vulnerability²

XSS is similar to SQLI. An SQLI attack targets the query function that interacts with the database whereas an XSS attack targets the HTML output function that sends data to the browser. When a user input is referenced as a data in a HTML page, an attacker may try to include HTML special characters such as the tag `<script>` which is able to invoke the JavaScript interpreter. If the application does not filter such special characters, the attacker's XSS injection is successful. Upon a successful attack, exploits such as account-hijacking, cookie-poisoning, denial-of-service, and web-content-manipulation can be performed. Typical input sources that attackers manipulate include HTML forms, cookies, URLs, and external files. JavaScript is often used by attackers but all kinds of client-side scripts such as VBScript and Flash could cause XSS.

Like SQLI, depending on the ways user inputs are referenced in HTML pages, XSS can be classified into three types: stored, reflected, and DOM-based. *Reflected* XSS works in a similar way as first order SQLI. It occurs when a server program references an unchecked data accessed from incoming HTTP request parameters in an immediate web page sent back to the user. This type of XSS is commonly found in error messages and search results. *Stored* XSS works in a similar way as second order SQLI. It occurs when a web server program stores an unrestricted user input in a persistent data store such as database and then the program accesses and references the stored data in a web page viewed by different users. This type of XSS is commonly found in forums, blogs, and other social networking sites. These two XSS-types result from improper handling of user inputs in server side scripts. By contrast, *DOM-based* XSS appears when a client side script itself references a user input dynamically obtained from the DOM (document object model) structure without proper validation and thus, malicious scripts injected via DOM-based XSS need not appear in server side scripts [71].

Regardless of the type of XSS, code injection techniques are similar. Using Figure 2-1 and Figure 2-2, the following discusses some of the commonly-found XSS vulnerabilities and the injection methods used by attackers to exploit those vulnerabilities.

Figure 2-1 shows a sample program for sharing tips about the places the users traveled. It contains four input fields—'Action', 'Place', 'Tip', 'User' that can be manipulated by attackers. The program can be called via a URL such as the one shown in Figure 2-2a.

² This section is reprinted, with permission, from L.K. Shar and H.B.K. Tan, "Defending against cross-site scripting attacks," *IEEE Computer*, March 2012. © 2012 IEEE


```

1 <html>
2 <title>Forum for Traveling Tips</title>
3 <body>
4 <h1>Welcome <script language="javascript" src="travelerInfo.js">
   </script>!</h1>

<%
5 String action = request.getParameter("Action");
6 String place = request.getParameter("Place");
7 if (place !=null && action.equals("Post")) {
8     String new_tip = request.getParameter("Tip");
9     if(new_tip.length < 100) {
10         stmt.executeUpdate("INSERT INTO forum VALUES ( " +
                               place + ", "+ new_tip + ")" );
11         out.println("Your Post has been added under Place '"
                       + HTMLEncode(place)+"'");
12     }
13     else {
14         out.println("Your Message: '" +new_tip+ "' is too long!");
15     }
16 }
17 else if (place !=null && action.equals("View")) {
18     ResultSet rs = stmt.executeQuery("SELECT * FROM forum
                                       WHERE place= "+place);
19     out.println("Here are the tips about visiting this place...");
20     while(rs.next()) {
21         String tip = rs.getString("tip");
22         out.println("'" +tip+"'");
23     }
24 }
25 }
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }
63 }
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 }
102 }
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }
125 }
126 }
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

(a) travelerTip.jsp

```

<SCRIPT>
24 var pos=document.URL.indexOf("User=")+5;
25 document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>

```

(b) travelerInfo.js

Figure 2-1. Sample server-side program and client-side script vulnerable to XSS.

In the program in Figure 2-1, statement at line 12 is vulnerable to reflected XSS due to the replay of invalid input supplied by users (line 8, 9, 12). An attacker might send a seemingly innocuous URL link like the one shown in Figure 2-2b to a victim via emails or social networking sites. The malicious script included in the link (shown in bold in Figure 2-2b) shall then be

executed on the victim's browser if the victim trusts the website 'travelingForum' and follows the link.

Statement at line 18 is vulnerable to stored XSS as the program stores user-supplied messages without proper sanitization (line 8-10) and displays them to visitors (line 14, 16-18). An attack can be carried out via a URL such as the one shown in Figure 2-2c.

Statement at line 4 is vulnerable to DOM-based XSS since the program includes a JavaScript file (Figure 2-1b) which accesses ‘User’ information from the URL (line 24) and displays it, without any sanitization, to users (line 25). Therefore, similar to the above reflected XSS scenario, the attackers may exploit this vulnerability using a crafted URL like the one shown in Figure 2-2d.

[illegible]

Figure 2-2. Example URLs that direct web users to travelingForum. (a) Ordinary URL that activates travelerTip.jsp in ‘View’ action. (b) URL that causes reflected XSS. It contains a malicious HTML Meta script capable of making a refresh request to travelingForum’s server every 0.3 seconds, potentially causing a denial-of-service. (c) URL that creates stored XSS scenario. It contains a JavaScript capable of sending client’s cookie information to a hacker’s website. (d) URL that causes DOM-based XSS. It contains a script capable of injecting misleading information over an original message.

2.2 Program Analysis

Program analysis³ techniques can be broadly categorized into two types—static program analysis and dynamic program analysis. In the following, we discuss static and dynamic program analysis techniques, such as control flow and data flow analysis and dynamic testing techniques, which shall be applied in our proposed approaches.

2.2.1 Static Program Analysis

Static program analysis techniques are techniques that infer properties of program points of interests without executing the program. Control flow analysis and data flow analysis are the two main static analysis techniques that are used to extract program properties such as control dependencies, data dependencies, infeasible paths, and anomalous behaviors. This information is essential in many software engineering applications such as code-based testing [72], code optimization [34], and program comprehension and debugging [105]. These two analyses are also the main techniques used in our proposed vulnerability mitigation approaches presented in this thesis. In our work, we use control flow and data flow analysis to extract control and data dependence relations of statements in *web programs*.

For terms and definitions related to control flow and data flow analysis, we use the ones given by Sinha et al. [140] throughout this thesis.

Control flow analysis [2] determines the relationships among program statements or the order in which program statements are executed. It identifies which statements control the execution of other statements. The control flow relationships analyzed from a given program are typically expressed using a directed graph called *control flow graph* (CFG).

As given by Sinha et al. [140], a CFG of a program P is a directed graph $G = (N, E)$ in which N contains a set of nodes and $E = \{(n, m) \mid n, m \in N\}$ contains an edge for each possible flow of control between the nodes. A node in a CFG of P could be a basic block representing an uninterrupted consecutive sequence of statements in P or it could represent a single program statement. In this thesis, we follow the latter representation. That is, each node in a CFG represents one statement in P . Thus, we may use ‘node’ and ‘statement’ interchangeably depending on the context.

³ Program analysis may be performed on source code or binary code. But in this thesis, program analysis is referred to as program analysis on source code because we do not consider binary code analysis in this thesis.

A program that consists of multiple procedures is typically expressed as an *interprocedural CFG* (ICFG), which consists of a set of inter-connected CFGs, one for each procedure. The construction method of ICFG is provided in detail by Sinha et al. [140], which is followed by our work. Throughout this thesis, we may use the term ‘CFG’ to also mean ‘ICFG’ as there is no difference in performing control flow and data flow analysis on ICFG or on CFG, apart from the additional construction step to inter-connect the CFGs. The definitions and terms defined in the followings are applicable to both CFG and ICFG.

A CFG has a unique entry node and a unique exit node. The exit node has no out-going edge. A *predicate node* in a CFG has two out-going edges, labeled with ‘true’ or ‘false’; and all other nodes have single out-going edge. The two out-going edges at a predicate node are called branches. Hence, they are ‘true’ branch and ‘false’ branch respectively. A *path* in a CFG is a sequence of nodes $\langle n_1, n_2, \dots, n_j \rangle$ where $(n_i, n_{i+1}) \in E$ for $1 \leq i \leq j-1$.

A node x in a CFG *dominates* a node y if and only if every path from the entry node to y in the CFG passes through x . A node x *postdominates* a node y if and only if every path from y to the exit node passes through x .

A node x is *control dependent* on a node y if and only if y has successors y' and y'' such that x postdominates y' but does not postdominate y'' . In general, if a node x is control dependent on a node y , then y must be a predicate node.

Typically, control dependency relation is expressed as a *control dependence graph* (CDG) in which nodes represent program statements and edges connecting the nodes represent control dependency relations between the statements.

Data flow analysis [3] is the process of extracting information about the possible set of values at program points of interests. It determines the locations of variable definitions that may reach a particular program point (*reaching definition*) and determines which variable definitions are ‘live’ at a particular program point, that is, which variable definitions given before that point are used after (*live variable*). It identifies a variable definition and all the variable uses reachable from that definition (*definition-use*). Inversely, it can also identify a variable use and variable definitions which can reach that use (*use-definition*).

A node x is *data dependent* on a node y if there exists a variable v such that y defines v , x uses v , and there is a path from y to x along which v is not redefined.

Typically, data dependency relation is expressed as a *data dependence graph* (DDG) in which nodes represent program statements and edges connecting the nodes represent data dependency relations between the statements.

For constructing CFG, CDG, and DDG, we follow the algorithms provided by Ferrante et al. [34], Lengauer and Tarjan [74], and Korel [72]. For constructing ICFG, we follow the algorithm provided by Sinha et al. [140]. Since CDG and DDG are constructed from control flow graph, if constructed from an ICFG, the resulting CDG and DDG are interprocedural; otherwise, they are intraprocedural [140].

2.2.2 Dynamic Program Analysis

Dynamic program analysis is the analysis of program by actually executing it. Dynamic program analysis has been used in many applications such as profiling [15], anomaly detection [13, 56], and test input generation [8, 160].

In principal, static program analysis techniques can examine the whole program source and verify program properties of all possible executions. As the analysis is not based on any particular input, its result is valid for all inputs. But it is prone to false alarms as it generally cannot distinguish possible and impossible program behaviors. By contrast, dynamic analysis is the analysis of program behaviors as a result of one or more executions with selected test inputs. Although the analysis is not exhaustive like static analysis, it provides more accurate information about the program behaviors that it could analyze simply because the behaviors are as a result of concrete program executions. Hence, when one wants to achieve more accuracy in addressing a particular issue, dynamic analysis can be used. However, dynamic program analysis may suffer from runtime overhead and it typically requires more complex analysis framework. It would be a trade-off between analysis cost and accuracy.

Our approaches presented in this thesis are mainly based on static program analysis techniques and data mining techniques. Dynamic analysis is used only in our vulnerability prediction work. Therefore, we shall only provide some background on the dynamic analysis technique that we used and a few major dynamic analysis methods that are commonly used in mitigating web application vulnerabilities.

Profiling and anomaly detection: Dynamic analysis is used in our vulnerability prediction study to better profile input validation and sanitization functions implemented in web applications. As dynamic analysis does not cover all program behaviors, choosing adequate test inputs that could induce interesting program behaviors (e.g., anomalous behaviors) is important to profile a

program or a function. Security specialists such as OWASP [107, 109] and RSnake [118, 119] provide a list of test inputs that could test the vulnerabilities of the program code. By simulating program functions with such attack inputs and analyzing their output behaviors, one could profile functional behaviors and detect anomalies.

There are also runtime-based anomaly detection techniques [13, 56]. These techniques generally instrument programs so that they could analyze the behaviors of instrumented code during runtime. They also deploy proxies to interrupt the incoming and/or outgoing HTTP data and analyze program behaviors.

Test input generation: In software testing, generating an adequate test suite is a difficult and important task. A dynamic analysis technique called concolic execution [125] can be used to automate this task. In our context, concolic execution is used to generate suitable test inputs for automated testing of SQLI and XSS vulnerabilities [68, 160].

Concolic execution technique is a combination of symbolic execution and actual execution with concrete inputs [125]. Symbolic execution [69] executes a program with symbolic values. Inputs to programs are represented with symbolic expressions, and these expressions are tracked and updated along a program path being executed. After a path is executed, it results in a path constraint which consists of conditions of symbolic expressions. By solving the path constraint using a constraint solver, the set of possible inputs (called solution pool) that could explore the path is generated. If the path contains security-sensitive program operations and there exist malicious attack inputs in the solution pool, then the vulnerability is detected. Although this concept is totally based on static analysis and is seems powerful, there are a few limitations. When a program contains many loops, symbolic execution suffers from path explosion since it is computationally expensive to symbolically execute all the paths exhaustively. Some of the paths may be left un-explored and thus, some of the vulnerabilities may be missed. Furthermore, a constraint solver may not be powerful enough to solve complex constraints. However, in concolic execution, in these situations, the program is instead executed with concrete inputs (often selected randomly). If an input that results in executing that path is found, an adequate input for testing the path is found. Still, concolic execution may suffer from path explosion if there are many such paths. Random selection of concrete inputs does not also guarantee that the path can be executed.

2.3 Data Mining

Han et al. [53] defined data mining as “*extracting or mining knowledge from large amounts of data*”. Data mining is typically used to extract interesting patterns (such as dissimilarities,

unknown patterns, and anomalies) from existing datasets so that one can infer about the properties or characteristics of the new or future datasets based on the information extracted. Consequently, data mining techniques have been extensively used in software quality analysis methods such as software defect prediction and vulnerability prediction. In our context, program source code is data and vulnerability information is the knowledge we wish to obtain. We are interested in the application of data mining activities to program source code so as to obtain its vulnerability information.

Typically, mining activities consist of:

1. Data preprocessing
2. Data analysis
3. Evaluation

In the following, we discuss the intelligent techniques corresponding to these data mining activities, which are used in our work.

2.3.1 Data Preprocessing

As discussed in Section 2.2, we shall use static and dynamic program analysis techniques to extract program properties from source code. However, the extracted data might be too raw to be mined. Specifically, when we extract interesting code patterns from program source code, the data distributions of our extracted code patterns might be arbitrary or skewed. Different types of code patterns that we extract may be defined on different scales. Some of the extracted data might be redundant or irrelevant. There may also be noises or outliers in the extracted data.

Mining such raw data will result in low-quality outputs. There are a number of data preprocessing techniques, such as data cleaning, data integration, data transformation, data reduction, to address these problems. In our work, we mainly use data transformation and data reduction to cleanse our data.

Data transformation is the process that converts data into a form appropriate for mining purposes [53]. *Min-max normalization* is one such data transformation technique, in which the data is scaled such that it falls within a specified range such as 0.0 to 1.0. The transformation is linear as it maps an original value to a value within the specified range. Thus, it preserves the relationships among the original data values, meaning we do not lose any information in min-max normalization. As it allows the data miners to work in a standardized data space instead of an arbitrary one, we use this technique to address the arbitrary data distribution problem.

Data reduction is the process that attempts to reduce available data such that data mining is efficient yet produce the same (or almost the same) analytical results [53]. In our work, we are not interested in efficiency as our data sizes are usually not large and thus, mining our data should be mostly practical. We are actually interested in particular data reduction techniques that could identify and remove redundant or irrelevant data dimensions so that data miners are not confused by those data. *Attribute subset selection* techniques such as information gain which finds most informative data dimensions and *dimensionality reduction* techniques such as principal component analysis [57] which identifies linearly uncorrelated data dimensions from the original ones are such techniques that we shall use to address the data redundancy and irrelevancy issues.

Data cleaning techniques such as data discretization can be used to filter noisy data or outliers. But, in our works, we did not use additional data preprocessing techniques to specifically address them because some of the learning algorithms (such as decision tree induction and neural networks) used by us have built-in capabilities to filter noisy data.

2.3.2 Data Analysis

Data analysis is the core activity in a data mining process. After the raw data is preprocessed, the cleansed data is ready to be analyzed to make some intelligent decisions. There are a few different data analysis methods to extract interesting data patterns. Some major methods are association rules mining, classification, cluster analysis, sequence data mining, and graph mining. In this thesis, classification and cluster analysis methods are used to build prediction models.

Classification: Classification is the technique that predicts the class label of a data instance based on the knowledge learned from other data instances with known class labels. It contains two main phases—training and testing. In the first phase, a classification algorithm, such as naïve bayes, decision tree induction, logistic regression, or neural networks (details of these algorithms can be found in Han et al. [53]), learns from a set of data instances denoted as training set. Each training instance is represented by an attribute vector of n -dimension. Some examples of attribute are lines of code, number of program operators, etc. Training instances have to be labeled with class information (in our case, vulnerable or non-vulnerable). In the second phase, the trained classifier (hence, *the learned predictor or the learned prediction model*) is used to predict the class labels of test data instances. By matching the actual class labels and the predicted ones of test instances, the performance of the prediction model is evaluated. If the model's accuracy is acceptable, then it is ready to be used for classification of future data instances.

Cluster analysis: In classification, labeled instances are required to train classifiers so that the trained classifiers are able to predict the class label of newly acquired instances. Hence, classification method is a type of *supervised learning*. However, sometimes, it may be hard to obtain labeled data instances. The number of available labeled instances may not be sufficient to train classifiers. In such cases, cluster analysis can be useful. It attempts to group instances (also represented with attribute vectors of n -dimension) into clusters such that instances within a cluster are very similar but are highly dissimilar to those from other clusters. This analysis is a type of *un-supervised learning* due to its training without labeled classes and also without predefined classes (e.g., vulnerable and non-vulnerable). This technique can be used for outlier detection as outliers having characteristics different from normal instances would be isolated from groups of normal instances. If vulnerabilities can be considered as outliers among normal program statements, cluster analysis can be used for vulnerability prediction as well.

Both types of techniques—classification and cluster analysis have advantages and disadvantages. Due to supervised training with labeled instances, classification methods would be highly accurate in detecting known or similar patterns whereas cluster analysis due to its un-supervised learning scheme could detect unknown patterns.

2.3.3 Evaluation

Once a prediction model is built using either supervised or un-supervised learning scheme, the next step is to evaluate how good the model is or whether the model is good enough for practical uses. Typically, a model evaluation consists of data sampling and performance measure computation.

Performance measures: Many research works such as [7, 41, 90, 91, 138], in software defect prediction and vulnerability prediction studies, use a typical confusion matrix [4] (shown in Figure 2-3) to assess the performance of prediction models. From the matrix, the following performance measures are usually derived:

- Probability of detection or recall (pd) = $tp / (tp + fn)$
- Probability of false alarm (pf) = $fp / (fp + tn)$
- Precision (pr) = $tp / (tp + fp)$
- Accuracy (acc) = $(tp + tn) / (tp + fp + fn + tn)$
- Balance (bal) = $1 - \sqrt{\frac{(0 - pf)^2 + (1 - pd)^2}{2}}$

		Actual	
		Vulnerable	Non-vulnerable
Prediction	Vulnerable	True positive (tp)	False positive (fp)
	Non-vulnerable	False negative (fn)	True negative (tn)

Figure 2-3. A typical confusion matrix

Pd measures how good our prediction model is in finding actual vulnerable sinks. Pr measures the actual vulnerable sinks that are correctly predicted in terms of a percentage of total number of sinks predicted as vulnerable. Pf is generally used to measure the cost of the model, that is, increasing pd or pr by tuning the prediction model may, on the other hand, cause more false alarms. In an ideal situation, pd should be close to 1 and pf should be close to 0. That is, the vulnerability prediction model neither misses actual vulnerabilities nor throws false alarms. Bal measures how close the prediction model is to this ideal case, that is, the higher the value of bal , the closer it is. Acc measures the number of times the model predicted correctly in terms of a percentage of total number of sinks. In this thesis, we assess the performance of our learned predictors based on these measures.

Data sampling: Random split-sampling, cross validation, and bootstrapping are common data sampling techniques for evaluating a model. Each of these techniques has been proved to be suitable for evaluating the accuracy of data miners [163]. For our study, we shall use *m-fold stratified cross validation* method which has also been used by many software defect and vulnerability prediction studies [41, 90, 91, 138, 141].

In *m-fold cross-validation*, the dataset is randomly divided into m partitions. The learner is trained on $m-1$ partitions and then tested on the remaining partition, iterating until every partition has been served as the test set once. This process is repeated m times to avoid data sampling bias. The performance measures are computed based on the overall numbers of correct and incorrect predictions from all the m iterations. In *stratified cross-validation*, the partitions are stratified so that the class distribution of the instances in each partition is approximately the same as that in the original un-partitioned data.

Chapter 3

SUPERVISED VULNERABILITY PREDICTION FROM STATIC CODE ATTRIBUTES⁴

Researchers have proposed sophisticated vulnerability detection approaches based on static and dynamic taint analysis techniques to address SQLI and XSS security risks. Though these taint analysis-based approaches have been shown to be effective at detecting many SQLI and XSS vulnerabilities, static approaches, such as [61, 82, 165], generally produce too many false alarms. Dynamic approaches, such as [9, 68, 160], are usually more accurate. But while dynamic analysis typically requires complex analysis frameworks such as concolic execution [68], full implementations of some of these approaches [9, 68, 160] are neither commercially nor publicly available as well. As a result, software development teams face difficulties in adopting these existing approaches. The growing numbers of vulnerability reports in security databases such as BugTraq [19] further support the need to find *alternative or complementary* solutions.

On the other hand, the applications of data mining techniques in software defect prediction and vulnerability prediction have recently been very popular. In this domain, researchers correlate software metrics (attributes) with defects [10, 92, 98, 100, 170] or vulnerabilities [42, 101, 138, 156]. There are also research works that compare the performances of different supervised learners based on a common set of software metrics [7, 75, 91].

In general, *supervised prediction models* categorize software modules/components/programs, represented by a set of software attributes, into one of the classes (e.g., vulnerable and non-vulnerable) by using classifiers that are trained on the same set of attributes obtained from software modules with known defect or vulnerability information [123]. The advantage of data mining techniques is that commonly-used *static code attributes* such as size, Halstead [52], and cyclomatic complexity [87] attributes can be easily collected. The availability of open source data

⁴ ACM, 2012. This chapter is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in L.K. Shar and H.B.K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *International Conference on Automated Software Engineering*, pp. 310-313, Sept. 2012. <http://doi.acm.org/10.1145/2351676.2351733>

mining tools such as the *Weka* tool described in Witten and Frank [163] also allows software engineers to readily apply those data mining methods.

Hence, prediction methods could provide a practical and efficient solution to mitigate web application security issues. As mentioned by Menzies et al. [91], these data miners may not accurately identify vulnerabilities like sophisticated vulnerability detection methods. But they would be useful at providing probabilistic remarks about the vulnerabilities of code sections. Software testers could then save time and effort by focusing on the code sections predicted to be vulnerable. However, there are still developments required to improve existing vulnerability prediction frameworks. One major issue is the granularity. To date, all existing frameworks [42, 101, 138, 156] locate vulnerabilities in a coarse-grained manner, that is, locate only vulnerable software modules or components. Much vulnerability auditing effort is still required to find and fix the specific, vulnerable code. Furthermore, some of those existing frameworks require process attributes (such as developer activity) to build vulnerability predictors [138]. Unlike static code attributes, process attributes are generally difficult to be collected, and the measurements are often inconsistent across projects [7, 92], affecting the accuracy of prediction. A vulnerability prediction framework that is both accurate and fine-grained would benefit software auditors.

Web applications in general adopt two types of methods, input validation and input sanitization, to prevent security vulnerabilities [108, 111]. An application is vulnerable if the implementation of these two methods is inadequate or there is no such method implemented. Consequently, the characteristics of input validation and sanitization implemented in a program could be useful for predicting the program's vulnerability.

The above observations serve as our main motivation for this work. In this chapter, we propose a set of static code attributes that characterize input validation and sanitization code patterns. Our aim is to build vulnerability prediction models from these attributes and available vulnerability information, which provide high recalls and low false alarm rates. Compared to current vulnerability prediction frameworks, we propose a novel framework that only requires collecting static code attributes and locates vulnerable code at statement level.

Contributions and Results

- We propose a novel set of static code attributes characterizing common input sanitization and input validation code patterns.
- We present a vulnerability prediction framework built upon the proposed static attributes.
- We also provide a prototype tool called *PhpMiner* that extracts the data of our proposed attributes from PHP programs.

- We evaluate three different SQLI and XSS vulnerability prediction models based on the experiments on a set of open source PHP-based web applications. We use statistical inference methods such as Wilcoxon signed-rank test to show that the results are statistically significant.
- In the experiments, our best XSS vulnerability predictor achieved, on average over 7 datasets, 84% recall and 15% false alarm rate. And our best SQLI vulnerability predictor achieved, on average over 4 datasets, 97% recall and 18% false alarm rate.

This chapter is organized as follows. Section 3.1 discusses our research hypothesis to be investigated in this study. Section 3.2 proposes the static code attributes that characterize input validation and sanitization code patterns. Section 3.3 presents our vulnerability prediction framework. Section 3.4 describes our prototype tool and the datasets collected for experiments. Section 3.5 evaluates the performance of vulnerability prediction models learned from our proposed attributes. Section 3.6 concludes this chapter.

3.1 Research Hypothesis

Both SQLI and XSS vulnerabilities arise from improper handling of inputs in web application programs. Hence, they are application-level vulnerabilities. In a typical web program, user inputs are accessed via forms, URLs, cookies, and XML files. Those inputs are often processed and propagated to various program points to accomplish the application's objectives. Some of those inputs may then be stored in the application's persistent data stores, such as databases and session objects, for further processing of the application's required functionalities. The operations carried out in those processes often include *security-sensitive program statements (sinks)* such as HTML outputs and database accesses. When user inputs referenced in such operations have not been sanitized or validated, vulnerabilities arise. XSS vulnerabilities arise when an unrestricted user input is used in an HTML output statement. SQLI vulnerabilities occur when a user input is used in an SQL statement without proper checks.

Hence, to prevent web application vulnerabilities, developers often employ input validation/sanitization methods along the paths propagating the inputs to the sinks. These methods can be broadly categorized into escaping, meta-character matching/removal, string length truncating, and data type checking/conversion. The methods typically employed are language-provided standard sanitization and validation functions (e.g., `htmlentities`) or custom functions developed for specific security or data integrity requirements by a developer or by a group of security experts.

From the analysis of many vulnerability reports in security databases such as CVE [27], we derived the following observations (also highlighted in detail in our background chapter 2.1):

- First, the vulnerability reports show that most of these vulnerabilities arise from the misidentification of inputs. That is, developers may implement adequate input validation and sanitization methods but yet, they may fail to recognize all the data that could be manipulated by external users, thereby missing some of the inputs for validation. Therefore, in security analysis, it is important to first identify all the inputs and the sinks that use them.
- Second, when an input to be used in a sink is considered to be a numeric type, the most effective defense method is to apply numeric-type check or numeric-type conversion (from string since inputs to web applications are, by default, strings).
- Third, we observed that in some cases, language-provided standard sanitization and validation functions are insufficient or inadequate to cleanse the inputs used. Hence, developers may also write their own piece of validation and sanitization code or adopt security functions developed by a group of security experts such as OWASP [110]. In those cases, such custom code usually comprises of a set of string functions and operations such as string replacement and string matching. These functions may also be regular expression-based.
- Last, different defense methods are generally required to prevent different types of vulnerabilities. For example, to prevent SQLI vulnerabilities, escaping characters that have special meaning to SQL parser is required whereas escaping characters that have special meaning to client script interpreters is needed to prevent XSS vulnerabilities. Thus, care must be taken to use appropriate methods.

The above observations lead us to our hypothesis (*H1*): Static code attributes that reflect commonly used input validation and sanitization code patterns could predict vulnerabilities.

From *H1*, in next section, we derive the static code attributes to build vulnerability predictors.

3.2 Static Code Attributes

Static code attributes that we propose are based on the control flow graph (CFG) of a web application program. Each node in the CFG represents a single program statement. Therefore, we shall use the terms ‘program statement’ and ‘node’ interchangeably depending on the context.

Our proposed approach presented in this chapter is limited to PHP programs. However, it is possible to extend our logic to other programming languages. SQLI and XSS vulnerabilities are widespread in PHP and Java-based applications. To extend our approach to Java, one can predefine Java-based functions according to our classification schemes described in this section. And since our approach only requires simple data flow analysis for data collection, it can be easily implemented using open source Java program analysis tools such as *Soot* [142].

3.2.1 Backward Static Program Slice

As inputs coming into web application programs are strings, input validation checks and sanitization operations performed in a program are mainly based on string operations. Therefore, our concept is to classify the natures of string operations applied, according to their potential effects on the tainted-ness of the variables referenced in a security-sensitive statement k . Intuitively, such validation checks and operations can be found in a *backward static slice* of the given web program with respect to k and the set of variables referenced in k . As given by Weiser [162], backward static slice with respect to *slicing criterion* $\langle k, V \rangle$ contains all nodes (including predicate nodes) in the CFG which may affect the values of V at k where V is the set of variables used in k . In this chapter, we shall use the notation S_k as the set of nodes resulting from a backward static slice with slicing criterion $\langle k, V \rangle$.

Hence, the first step of our method is to classify the nodes in S_k according to their security-related properties, and then to capture these classifications in a set of attributes on which vulnerability predictors are to be built. Basically, our approach attempts to answer the following research question: “*Given a program slice of a sink, from the number of inputs, and the numbers and types of input validation and sanitization functions found on the nodes in the slice, can we predict the sink’s vulnerability?*”

For classification, we perform static program analysis on S_k . That is, classification is carried out via static checking of the properties of the functions or operations performed in each node n in S_k . That is, from the language built-in functions that have specific security purposes (e.g., `addslashes`), the language operators (e.g., string concatenation operator `.`), or the predefined language parameters (e.g., `$_GET`) used, n is classified statically. As a control flow node n may contain a variety of program operations, there may be multiple classifications for n (see example in Section 3.2.3). The static code attributes that characterize such functions and operations are listed in Table 3-1 and presented in the following sub-section.

3.2.2 Classification

Input and sink classification: We call a node u in a CFG at which the data submitted by an external user is accessed an *input node*. The nodes which reference data from HTTP request parameters, database, and XML files are some examples of input nodes. According to different natures of input sources, we classify inputs into the following types (attributes 1-7 in Table 3-1):

- 1) *Client*: Data submitted via HTML forms and URLs (e.g., `$_GET`, `$_POST`).
- 2) *File*: Data accessed from external files such as cookies and XML files (e.g., `$_COOKIE`, `fgets`). The contents in these files may have been tempered by malicious users.
- 3) *Database*: Data retrieved from database that is updated by programs with unknown vulnerabilities (e.g., `mysql_fetch_array`). As the data retrieved from database could be of different data types, we further classify the database inputs into two sub-types—*Text* and *Other*. This is to reflect the fact that string data from database are often exploited to cause security attacks such as second-order SQLI and stored XSS. Therefore, *Text-database* inputs represent ‘String’ type data (e.g., `varchar`, `text`, and `blob`). *Other-database* inputs represent data of any other data types such as ‘Numeric’ (e.g., `int`) and ‘Date and Time’ (e.g., `timestamp`).
- 4) *Session*: Data accessed from persistent data objects that may have been defined by programs with unknown vulnerabilities (e.g., `$_SESSION`).
- 5) *Uninit*: Variables which may not have been initialized in the program. In programming languages such as PHP, variables which have not been defined in the program are often referenced in sensitive program operations because it is possible to configure settings such as `register_global` so that any un-initialized variable is automatically assigned by data from HTTP request parameters.

It is clear that ‘*Client*’, ‘*File*’, and ‘*Uninit*’ are inputs that would definitely cause SQLI and XSS vulnerabilities if used in the sinks directly without any checks. ‘*Database*’ and ‘*Session*’ represent data stored in usually different programs and hence, their tainted-ness is usually unknown or hard to be tracked. A naïve taint analysis-based vulnerability detection approach (such as [61], [82], or [157]) might result in many false negatives if users assume that such inputs are safe and in false positives if users assume otherwise. This is the area where our proposed vulnerability predictors are expected to perform better because the prediction is not based on user assumptions but instead based on historical information.

Table 3-1. Static code attributes

Attribute ID	Attribute Name	Description	Data Type
1	Client	The number of nodes that access data from HTTP request parameters	Numeric
2	File	The number of nodes that access data from files	Numeric
3	Database	The number of nodes that access data from database	Numeric
4	Text-database	Boolean value 'TRUE' if there is any text-based data accessed from database; 'FALSE' otherwise	Boolean
5	Other-database	Boolean value 'TRUE' if there is any data except text-based data accessed from database; 'FALSE' otherwise	Boolean
6	Session	The number of nodes that access data from persistent data objects	Numeric
7	Uninit	The number of nodes that reference un-initialized program variable	Numeric
8	SQL	The number of SQL sink nodes	Numeric
9	HTML	The number of HTML sink nodes	Numeric
10	Null	The number of nodes that perform data presence check	Numeric
11	Size	The number of nodes that perform data size or length check	Numeric
12	Containment	The number of nodes that perform data containment check	Numeric
13	Match	The number of nodes that perform string-based matching	Numeric
14	Regex-match	The number of nodes that perform regular expression-based matching	Numeric
15	Numeric-type	The number of nodes that perform numeric data type check	Numeric
16	Other-type	The number of nodes that perform type check excluding numeric data type	Numeric
17	SQLi-sanitization	The number of nodes that apply language-provided sanitization functions for preventing SQLi issues	Numeric
18	XSS-sanitization	The number of nodes that apply language-provided sanitization functions for preventing XSS issues	Numeric
19	Encoding	The number of nodes that encode data	Numeric
20	Encryption	The number of nodes that encrypt data	Numeric
21	Replacement	The number of nodes that perform string-based substring replacement	Numeric
22	Regex-replacement	The number of nodes that perform regular expression-based substring replacement	Numeric
23	Numeric-conversion	The number of nodes that convert data into a numerical format	Numeric
24	Un-taint	The number of nodes that return predefined information or information not influenced by external users	Numeric
25	Propagate	The number of nodes that propagate the tainted-ness of an input string	Numeric
26	Custom	The number of user-defined functions	Numeric
27	Other	The number of nodes that are not classified as any of the above types	Numeric
Target attribute			
28	Vulnerable?	Target attribute which indicates a class label—Vulnerable or Non-Vulnerable	Boolean

We call a node k in the CFG of a web program a *sensitive sink* if the execution of k may lead to harmful operations. Throughout the chapter, we shall address k as a sensitive sink. This study relates to two types of sensitive sinks (attributes 8-9 in Table 3-1):

- 1) *SQL*: Database operations that are susceptible to SQLi attacks (e.g., `mysql_query`).
- 2) *HTML*: HTML output operations that are susceptible to XSS attacks (e.g., `print` and `echo`).

Input validation and sanitization classification: To prevent SQLi and XSS vulnerabilities, web applications generally adopt various input validation and input sanitization routines. Input validation ensures that user inputs conform to a required input format. Input sanitization removes,

replaces or escapes malicious characters from user inputs according to the context of a sensitive program operation so that those characters may not cause the program to perform unintended operations. As such, whether or not a sensitive program operation is vulnerable is dependent on the effectiveness of validation checks and sanitization operations applied with respect to the context of that particular operation. In consequence, static code attributes that reflect the characteristics of these methods could be used to predict vulnerability.

A variety of input validation checks and sanitization operations may be found in the nodes in S_k . For example, a node (or a set of nodes) in S_k may check if an input representing a client's age contains only numeric values. Another node may remove some predefined characters from an input. Different operations may serve different purposes and may have different effects on the tainted-ness of an input. Hence, we shall classify the nodes in S_k such that the classifications reflect the potential input validation and sanitization operations performed on k . Each node in S_k may be classified into one or more types depending on its properties (i.e., functions invoked or operators used). The classification scheme is as follows (attributes 10-24 in Table 3-1):

- 1) *Null*: functions that check if the data is present (e.g., `isset`).
- 2) *Size*: functions that return the size or length of an argument (e.g., `strlen`).
- 3) *Containment*: functions that check if an argument contains any predefined characters (e.g., `strpos`).
- 4) *Match*: functions or operations that compare an argument with another argument (e.g., `strcmp`, `$a == "abc"`).
- 5) *Regex-match*: regular expression-based string matching functions (e.g., `preg_match`).
- 6) *Type*: functions that check if the type or format of an argument. We further classify this check into *Numeric-type* check (e.g., `is_int`) and *Other-type* check (e.g., `is_string`, `is_file`) to reflect our experience that numeric data type checks usually un-taint the input values.
- 7) *SQLI-sanitization*: Functions designed to prevent SQL injection issues (e.g., `addslashes`, `mysql_real_escape_string`). Functions that invoke stored procedures or prepared statements (e.g., `$query->prepare`) are also classified as *SQLI sanitization* type.
- 8) *XSS-sanitization*: Functions designed to prevent XSS issues (e.g., `htmlspecialchars`).
- 9) *Encoding*: Functions that encode an argument according to a specific encoding format (e.g., `urlencode`). An input variable may be properly sanitized using encoding functions (e.g., `<a href= 'login.php?name=' .urlencode(input)>`).
- 10) *Encryption*: Encryption or hashing functions designed to ensure secure data transfer (e.g., `crypt`).

- 11) *Replacement*: String-based substring replacement functions (e.g., `substr_replace`).
- 12) *Regex-replacement*: Regular expression-based substring replacement functions (e.g., `preg_replace`).
- 13) *Numeric-conversion*: Functions that process one or more arguments and return a numeric value (e.g., `floatval`) or numeric type casting operations (e.g., `$x = (int) $_GET['id']`).
- 14) *Un-taint*: Functions or operations that return predefined information (e.g., `$a='text'`), information derived from configuration settings (e.g., `localeconv`), or numeric information derived from program operations (e.g., `mysql_field_len`). These functions or operations may not be intended as input validation or input sanitization but are considered as one because the resulting data is generally benign.

Clearly, nodes in S_k may also include ordinary operations that may or may not serve any security purpose. They may simply *propagate* the input. Furthermore, functions invoked at n could also be *customized* library functions. From the empirical studies, we observed that the above input validation and input sanitization types are commonly implemented to prevent web vulnerabilities. There may also be *other* types of preventive measures that we did not observe. Hence, any other node that does not belong to any of the above types shall be considered as *other*. The following classification scheme is defined for such nodes:

- 1) *Propagate*: functions or operations that may convert arguments into different representations but return part or whole of the original arguments (e.g., `substr`, `trim`, `explode`, `$str=$str.'abc'`); functions that unquote or decode arguments (e.g., `html_entity_decode`, `urldecode`, `stripslashes`).
- 2) *Custom*: library functions with unknown vulnerabilities.
- 3) *Other*: functions or operations that are not classified as any of the above types.

Target attribute: The last attribute *Vulnerable?* in Table 3-1 is the target attribute which is used to indicate the class label to be predicted.

Attribute vector: As we are predicting the vulnerability of the sink, every sink is an individual, independent instance that provides input to a data mining scheme. As we propose 28 static code attributes, every instance is characterized by a 28-dimensional attribute vector. The value of each attribute in that vector is a measurement of the quantity to which the attribute refers. For example, if there are five inputs accessed from database in the slice of a sink, the value of the attribute '*Database*' is five. As shown in Table 3-1, 25 attributes are numeric and 3 attributes are Booleans.

3.2.3 Example

In this section, we illustrate the classifications of a few sample program statements and the attribute collection process using the PHP program in Figure 3-1, extracted from a vulnerable application called *PhpNuke* (its vulnerability report can be found in BugTraq 10493 [19]).

In Figure 3-1a, the program statements 16, 19, 21, 22, and 23 are *HTML* sinks. Statements 16 and 22 are non-vulnerable *HTML* sinks and statements 19, 21, and 23 are vulnerable *HTML* sinks. Variables such as `$title`, `$url`, `$url_title`, and `$id` are *Client* inputs as these variables may be defined by HTTP GET, POST, or COOKIE parameters because of the command in statement 2.

Likewise, the rest of the statements in the program can be classified as follows. Statement 4-8 shall be classified as *XSS-sanitization* type because the statements invoke language-provided XSS sanitization function. Statement 9 is *Null* type as it checks whether an input data is present. Statement 10 and 14 are *Un-taint* type as they define a variable with a predefined value. Statement 11 is *Regex-match* type as it uses regular expression matching to check if an input data conforms to a valid URL format. Statement 12 is *Propagate* type as it only performs string concatenation. Statement 13 is *Numeric-type* type as it validates a variable to be numeric. Statement 15 and 20 are *Match* type as they compare a variable to a value. Statement 18 is *Propagate* type as the `urldecode` function it invokes only returns the decoded value from a variable.

It is clear that the above classifications can be carried out by checking the language built-in functions invoked, the parameters accessed, or the operator used. One has to predefine the classification types of language built-in functions such as `urldecode`, parameters such as `$_GET`, and operators such as `'.'` in a database.

Figure 3-1b shows the sub-CFG of the program in Figure 3-1a obtained by backward slicing with the slicing criterion $\langle 21, \{ \$url, \$url_title \} \rangle$. Like *HTML* sink 21, backward program slices with respect to *HTML* sinks 16, 19, 22, and 23 can also be computed, which would result in slices containing the nodes {3, 6, 9, 10, 13, 14, 15, 16}, {4, 6, 9, 10, 13, 14, 15, 18, 19}, {5, 6, 9, 10, 13, 14, 15, 22}, and {9, 10, 13, 14, 15, 23} respectively.

```

1  <?php
2  import_request_variables('GPC');
3  $errmsg = "invalid input";//un-taint

4  $title = htmlspecialchars($title);//xss-sanit
5  $text = htmlspecialchars($text);
6  $reviewer = htmlspecialchars($reviewer);
7  $url = htmlspecialchars($url);//xss-sanit
8  $url_title = htmlspecialchars($url_title);
9  if($reviewer == null) //null
10   $error = 1; //un-taint
11  if (!ereg('(^http[s]*:[/]+)(.*)', $url)) //regex-match
12   $url = "http://" . $url;//propagate
13  if (!is_int($score)) //numeric-type
14   $error = 1; //un-taint
15  if ($error == 1) { //match
16   echo $errmsg;//non-vuln html
17  } else {
18   $title = urldecode($title);//propagate

19   echo "<center><b>REVIEWCOMMENT: $title </b></center><form
        method='post' action='modules.php?name=review'>"//vuln html
20   if ($url != "")//match
21     echo "<b>RELATEDLINK:</b><a href='$url'>$url_title</a><br>"//vuln html

22   echo "<blockquote><p>$text<br>
        <b>REVIEWER: $reviewer</b><br>
        <b>SCORE: $score</b><br>"//not-vuln html

23   echo "<input type='hidden' name='id' value=$id>
        <input type='submit' name='rop' value='YES'>
        </form>"//vuln html
  }

```

Figure 3-1. (a) Sample Web application program that contains vulnerable and non vulnerable sinks (modified code snippet from PhpNuke/Review/index.php downloaded from [143]). Statement 19, 21, 23 are vulnerable *HTML* sinks and statement 16 and 22 are non-vulnerable *HTML* sinks. (b) Control flow graph representing the backward program slice of statement 21.

Hence, according to our classification schemes, the attribute vectors for *HTML* sinks 16, 19, 21, 22, 23 extracted from their respective backward program slices would be represented as shown in the following:

Attribute sinks	Client	Null	Match	Regex- match	Numeric -type	XSS- sanitization	Pro- pagate	Un- taint	...	Vulner- able?
16	1	1	1	0	1	1	0	3	...	No
19	1	1	1	0	1	2	1	2	...	Yes
21	1	1	2	1	1	2	1	2	...	Yes
22	1	1	1	0	1	2	0	2	...	No
23	1	1	1	0	1	0	0	2	...	Yes

In the above table, the “...” refers to the rest of the attributes listed in Table 3-1, not shown here due to space limit. Their values would be either numeric or Boolean. Every sink would be represented by a 28-dimensional attribute vector like the ones above. In Chapter 4, we further illustrate the outputs of a classification model and a clustering model, mining on such attribute vectors as we introduce clustering model in Chapter 4.

3.3 Vulnerability Prediction Framework

Data preprocessing: To generalize the results, our vulnerability predictors must be able to handle data of arbitrary distributions. Excluding the target attribute, we have 27 static attributes. Twenty-five attributes take on numeric values and two attributes are binary. From our preliminary tests, we observed that different numeric attributes are defined on different scales and most of the attributes’ distributions are highly skewed. This may cause bias toward some attributes (e.g., attributes with large scale values), especially in the context of clustering where similarity measurement combines multiple attribute scales (cluster analysis is discussed in Chapter 4). We use a data standardizing technique called *min-max normalization* to avoid this problem, which is described in Han et al. [53].

As explained in our background chapter, Chapter 2.3, min-max normalization enables our predictors to work in a standardized data space instead of a raw data space. An attribute is normalized when its value is scaled so as to fall within a small specified range (we use the range of zero to one). As the normalized value is a linear transformation from the original data value, the relationships among the original data values are preserved. The min-max normalization is to be made for all the instances of every numeric attribute. This shall result in a set of values within the range of zero to one. The binary attributes do not need to be transformed.

Data reduction: When proposing a set of attributes that characterize the quality of software, it is important to identify the attributes that best capture it. This process is known as *feature ranking* or *attribute ranking*. It can also be used to identify irrelevant or redundant attributes. Hall and Holmes [51] reported that removal of such non-informative data may improve the performance of some classifiers.

Various attribute ranking techniques, such as chi-square, information gain, gain ratio, and symmetrical uncertainty, have been investigated in literature [41, 51]. These techniques independently rank attributes regardless of the classifier used. In this chapter, gain ratio (GR) method is used to find which attributes contributed most to the classifiers' performance and check if there are redundant attributes that degrade the classifiers' performance. This method is based on the concept of information theory. It constructs decision trees by choosing the attributes with most information as branches. Gain ratio for each attribute can be computed as follows (given by Han et al. [53]):

Given a dataset D in which there are n instances, the amount of information needed to classify an instance in D is:

$$H(C) = -(n_{vul} / n) \log_2(n_{vul} / n) - (n_{non-vul} / n) \log_2(n_{non-vul} / n),$$

where n_{vul} and $n_{non-vul}$ are the numbers of instances among n that belong to class *Vulnerable* and class *Non-Vulnerable* respectively. Then, the amount of information needed to classify an instance in D given the partitions of instances based on the distinct values of an attribute A is:

$$H(C | A) = -\sum_{j=1}^v (n_j / n) * [(n_{j_{vul}} / n_j) \log_2(n_{j_{vul}} / n_j) + (n_{j_{not-vul}} / n_j) \log_2(n_{j_{not-vul}} / n_j)],$$

where n_j is the number of instances that have j^{th} distinct value of A . $n_{j_{vul}}$ and $n_{j_{not-vul}}$ are the numbers of instances among n_j that belong to class *Vulnerable* and class *Non-Vulnerable* respectively. Thereby, information gain is defined as the change in information requirement after partitioning on A . That is:

$$IG(A) = H(C) - H(C | A),$$

Next, by applying a normalization method to information gain, gain ratio can be computed as:

$$GR(A) = \frac{IG(A)}{-\sum_{j=1}^v (n_j / n) * \log_2(n_j / n)}.$$

Classifiers: There are a number of different classification algorithms such as tree-based approaches, neural networks, support vector machines, nearest-neighbor approaches, statistical procedures, and ensembles. Literature studies [7, 91] have shown that different classification algorithms may produce different performances. In this study, we use *three different classifiers*—C4.5, Naïve Bayes (NB), and Multi-Layer Perceptron (MLP), to build SQLI and XSS vulnerability predictors.

The reason for selecting these classifiers is that this study does not aim at achieving good prediction accuracies through the use of advanced classifiers. Rather, it aims to show that the proposed attributes can predict vulnerabilities in general, regardless of the type of classifier used. Therefore, we selected the classifiers that are standard and commonly used in literature [75, 91]. Their implementations are also readily available in *Weka* [163]. Different types of classifiers are used so that we can prove the predictive capability of our proposed attributes across classifiers. We shall show in Section 3.5 that our attributes can predict vulnerabilities using a simple classifier like Naïve Bayes, which is dependent on the naïve assumption that the attributes are independent. Naïve Bayes is useful as a baseline learning method for evaluating new set of attributes [163]. Therefore, sophisticated learning methods like ensembles can only further improve the performance. But, further improvements using advanced data mining algorithms are for future work.

C4.5 is a decision tree-based classifier that recursively partitions the training data by means of attribute splits at each node of the tree [115]. The splitting criterion is the information gain that results from choosing an attribute for splitting the data. Information gain is computed using the same equations shown above. To build a C4.5 model, we use the default parameter values set in *Weka*, which are:

```
confidenceFactor = 0.25;  
minNumObj = 2;  
numFolds = 3
```

The parameter *confidenceFactor* is used for pruning (smaller values incur more pruning). The parameter *minNumObj* defines the minimum number of instances per leaf. The parameter *numFolds* determines the amount of data used for reduced-error pruning. One fold is used for pruning, the rest for growing the tree.

To illustrate, C4.5 executing on the five sample attribute vectors collected from the slices of *HTML* sinks in Figure 3-1 (discussed in Section 3.2.3) results in a tree:

```
Propagate ≤ 0: Non-Vulnerable  
Propagate > 0: Vulnerable
```


which can be interpreted as “a sensitive sink is vulnerable if the number of nodes classified as *Propagate* is more than 0”.

NB is a simple statistical classifier that estimates the posterior probability of each class of the target attribute (in our case, *Vulnerable* or *Non-Vulnerable*) based on the values of the training data so that a given module (in our case, a module is a sensitive sink) can be assigned to the class label with the highest probability. It follows Bayes’ theorem with the assumption that all attributes are conditionally independent. Bayes’ theorem is (given by Han et al. [53]):

$$P(H / X) = \frac{P(H)P(X / H)}{P(X)}.$$

where H is the hypothesis that an instance X is vulnerable (or not vulnerable). $P(H / X)$ is the posterior probability that X is vulnerable (or not vulnerable) given the information of X ’s attributes. $P(H)$ is the prior probability that any given instance is vulnerable (or non-vulnerable) regardless of its attribute information. $P(X)$ is the prior probability that any instance selected from a set of training instances has the same attribute information as X . $P(X / H)$ is the likelihood that a given instance has the same attribute information as X given that the instance is vulnerable (or not vulnerable). Like a C4.5 model above, to build a NB model, we use the default parameter values set in *Weka*, which are:

```
useKernelEstimator = False;
useSupervisedDiscretization = False;
```

Setting the parameter “useKernelEstimator” to ‘True’ would allow the NB model to use a kernel estimator for numeric attributes. But we use the default setting ‘False’ using instead a normal distribution. Likewise, setting the parameter “useSupervisedDiscretization” to ‘True’ would allow the NB model to use supervised discretization to convert numeric attributes to nominal ones. But again we use the default setting ‘False’.

MLP is a sophisticated classifier which depicts the neural network structure of the human brain:

As shown above, an MLP consists of an input layer, one or more hidden layers, and an output layer. The attribute data of training instances are fed to the units in the input layer. Weighting, aggregation, and thresholding functions are then iteratively applied to the data propagated along the units in the layers to predict the class label of an instance which is presented in the output layer. Like the above models, to build a MLP model, we use the default parameter values defined in *Weka*, which are:

```
No. of hidden layer = 1;
No. of neurons per hidden layer = (no. of attributes + no. of classes)/2
```

Default settings are used since we are not interested in optimizing any particular classifier. Rather, our focus is to endorse the predictive capability of our proposed attributes using different types of classifiers with default or commonly-used settings. Naturally, our future research direction could well be to focus on such optimization works. Detailed information of these classification algorithms can be found in standard data mining books such as Han et al. [53] and Witten and Frank [163].

Model Training and Testing: We use stratified 5-fold cross validation for training and testing the selected classifiers. Cross validation method is used by many prediction studies [41, 90, 91, 138]. Studies recommend 10-fold due to its relatively low bias and variance [53, 163]. But, due to the small size of one of our datasets, we resort to 5-fold validation.

The data is randomly divided into five sets. A classifier is trained on four sets and then tested on the remaining set; iterating until each set has been served as the test set once. This process is repeated five times. The order of training and test set is randomized. As explained by Menzies et al. [91], isolating a test set from the training set conforms to hold-out test design which is important to evaluate the classifier's capability to predict future vulnerabilities. This test design also overcomes the ordering effects [35] due to randomization. This is important to avoid a malignant increase in performance by a certain ordering of training and test data.

Performance Measures: To evaluate the vulnerability prediction models, we compute recall or probability of detection (*pd*), probability of false alarm (*pf*), precision (*pr*), accuracy (*acc*), and balance (*bal*). The definitions and equations of these measures are provided in our background chapter, Chapter 2.3.3. In an ideal situation, *pd* should be close to 1 and *pf* should be close to 0 such that the model neither misses many vulnerabilities nor throws many false alarms.

3.4 Prototype Tool and Dataset

This study concerns with two types of vulnerability prediction models—SQLI vulnerability prediction models and XSS vulnerability prediction models. However, as the data of our proposed attributes can be collected for any types of sinks, predictors for other application-level security vulnerabilities could be built as well.

The attribute vectors for *HTML* sinks and *SQL* sinks are collected from seven open source PHP-based web applications of varying sizes—*GeccBBLite*, *SchoolMate*, *FaqForge*, *WebChess*, *Utopia News Pro*, *Yapig*, and *PhpMyAdmin*, which can be downloaded from SourceForge [143]. All these benchmark applications have been used in evaluating major vulnerability detection approaches [61, 68, 160, 165]. Table 3-2 shows the information of these test subjects. It shows the PHP files from which the data of our proposed attributes are collected. It also shows the lines of code (LOC) of each test file. As shown in the last column of Table 3-2, their vulnerability information is known to public and accessible in various security advisories such as BugTraq [19].

For each sensitive sink in each PHP file, one attribute vector was collected. Every vector contains the data of our proposed attributes shown in Table 3-1. In the following, we explain the details of the method used to collect the data.

Since the data are to be collected from PHP-based programs, we implemented a data collection tool called *PhpMiner*. Its architecture is shown in Figure 3-2. Our prototype tool is based on an open source PHP code analysis tool called *Pixy* [61]. *Pixy* is a fully automated tool that tracks taint information flow in PHP programs. Given a PHP program, it generates control flow graph and provides data flow analysis of sensitive program points in the program. *Slicer module* uses *Pixy*'s definition/use APIs to compute backward static program slices of sensitive sinks according to the slicing algorithm given by Weiser [162]. Then, *Classifier module* classifies the nodes contained in the slices by statically analyzing the functions invoked or the operators used. *PhpMiner* handles over 300 PHP built-in functions and 30 PHP operators for classification. *Attribute Collector module* stores all the classifications made and finally, *PhpMiner* outputs an attribute vector for every sink.

Since *Pixy* identifies inputs and sensitive sinks based on the configuration files, in *Pixy*'s configuration, we defined the input sources of *Client*, *File*, *Database*, *Session*, and *Uninit* (un-initialized variables) so that *Pixy* could identify the input types classified by us. However, in order to differentiate the two sub-types of *Database* inputs—*Text* and *Other*, we needed to implement a database-schema analysis and a data flow analysis of *Database* inputs. Database-schema analysis

(database schema of the test subject is required) is carried out first to provide *PhpMiner* with the classifications of *Text* and *Other* database table columns. Thereafter, whenever *PhpMiner* finds *Database* inputs in the PHP program under test, it uses data flow analysis to verify the column name or the column index of the database table being accessed. It then uses information provided by database-schema analysis to classify the input type. We also configured in *Pixy* the two types of sensitive sinks: *SQL* sink and *HTML* sink. These analyses and functionalities are independently implemented in *Input and Sink Identifier* module. The configurable nature of *Pixy* allows us to define various types of sinks that are sensitive to different type of security vulnerabilities; and thus, *PhpMiner* could support a variety of vulnerability predictors.

Figure 3-2. Architecture of the prototype tool, *PhpMiner*

Hence, like size and code complexity attributes [87], our proposed attributes can also be easily collected through an automated tool. As our technique only requires definition/use analysis on CFG, any other program analysis tool could also be used. Table 3-3 shows the 11 datasets collected by *PhpMiner* using a Pentium 3.4GHz 4GBRAM Windows XP machine. The first seven datasets are collected from slices of *HTML* sinks and the last four datasets are collected from slices of *SQL* sinks (we did not use datasets of *SQL* sinks for some applications as we do not have sufficient data labeled with SQLI vulnerability information). The complete package of collected

datasets, the detailed workings of *PhpMiner*, and the PHP built-in functions handled by the tool are provided in the author's website [116].

Table 3-2. Statistics of the test subjects used

Test Subject	Description	Test File	LOC	Security Advisories
GeccBBLite 0.1	A simple bulletin board	index.php leggi.php rispondi.php scrivi.php	93 121 55 69	BugTraq-35449
SchoolMate 1.5.4	A tool for school administration	index.php	8145	groups.csail.mit.edu/ pag/ardilla/
FaqForge 1.3.2	Document creation and management tool	index.php admin\index.php	790 1448	Bugtraq-43897
WebChess 0.9.0	Online chess game	mainmenu.php chess.php	2205 3236	Bugtraq-43895
Utopia News Pro 1.1.4	News management system	index.php login.php postnews.php users.php	1294 1174 1570 1699	BugTraq-15028
Yapig 0.95b	Image gallery	view.php	4748	BugTraq-413255
PhpMyAdmin 2.6.0-pl2	A tool for handling MySQL database operations	select_server.lib.php left.php	89 952	PMASA-2005-01, PMASA-2005-05

Table 3-3. Datasets

Dataset	#HTML sinks	#Vuln. Sinks (%Vuln.)
geccbblite-html	20	10 (50%)
schoolmate-html	172	138 (80%)
faqforge-html	115	53 (46%)
webchess-html	73	22 (30%)
utopia-html	74	17 (23%)
yapig-html	21	6 (29%)
myadmin-html	58	16 (28%)
Dataset	#SQL sinks	#Vuln. Sinks (%Vuln.)
geccbblite-sql	9	4 (44%)
schoolmate-sql	189	152 (80%)
faqforge-sql	42	17 (40%)
webchess-sql	53	24 (45%)

3.5 Evaluation

This section investigates our research hypothesis (*H1*) that our proposed static code attributes can predict vulnerabilities.

The three selected classifiers are implemented in *Weka* [163]. To evaluate our vulnerability prediction framework, we configured the desired classifier and the data sampling mode (5-fold cross validation) in *Weka*, provided the dataset, and ran *Weka* on a Pentium 3.4GHz 4GBRAM

Windows XP machine. Each classifier was run on the datasets of *HTML* sinks and *SQL* sinks. Both C4.5 and NB classifiers took less than a second to complete each run whereas MLP classifier took nearly a minute to complete each run. For each run, *Weka* outputs the prediction results in a confusion matrix like the one shown in Figure 2-3. Performance measures are computed from each confusion matrix. Figure 3-3a shows the performance measures of each classifier run on the datasets of *HTML* sinks. Figure 3-3b shows the performance measures of each classifier run on the datasets of *SQL* sinks. The results are discussed in the following sub-section.

Note that in this study, we do not intend to compare the performances of different types of classifiers; instead we focus on how useful the proposed static code attributes are for predicting web application vulnerabilities. Therefore, we did not compare the performance among classifiers using suitable statistical tests such as Friedman test suggested by Demšar [30].

3.5.1 Result and Discussion

Overall, our vulnerability predictors achieved outstanding results. As shown in Figure 3-3, the MLP model was our best predictor based on the average results. For predicting XSS vulnerabilities, on average, the MLP model achieved $pd = 84\%$ and $pf = 15\%$. For predicting SQLI vulnerabilities, on average, it achieved $pd = 97\%$ and $pf = 18\%$. We observed that in the latter case, the MLP model performed better in terms of pd ; on the other hand, it also produced higher false alarm rates (3% higher) than the former case. The C4.5 model also produced good results on both XSS and SQLI vulnerability predictions. Although the NB model performed the worst among the three models, it still detected $\sim 80\%$ of SQLI vulnerabilities and $\sim 70\%$ of XSS vulnerabilities. Its relatively high false alarm rates could be due to some redundant attributes or some relations that might exist among the proposed static code attributes. Both of these issues are known to hurt NB classifier's performance because of its strong independence assumptions [163].

In recent defect prediction and vulnerability prediction studies [75, 90, 91, 92, 101, 138, 149], predictors with the results of ($pd \geq 70\%$ and $pf \leq 25\%$) have generally been benchmarked. Therefore, to provide a comparison point, we used Wilcoxon signed-rank test to compare the results of our best predictor (MLP) with a benchmark result ($pd, pf = (71\%, 25\%)$). This statistical method is suitable for pair-wise comparison of prediction models because it does not assume normal distributions and is less susceptible to outliers due to the use of ranked performances rather than performance means [96]. In terms of pd , our best predictor achieved statistically better results than the benchmark pd result of 71% (significant at 99%). However, in terms of pf , our predictor

was neither statistically better nor worse. This is mainly due to the high *pf* resulting from the test on *geccbblite-html*.

Measure (%)		<i>Pd</i>	<i>Pf</i>	<i>Pr</i>	<i>Acc</i>
Data & Classifier					
geccbblite-html	NB	70	50	58	60
	C4.5	60	40	60	60
	MLP	70	50	58	60
schoolmate-html	NB	89	53	87	81
	C4.5	96	18	96	93
	MLP	95	15	96	93
faqforge-html	NB	93	21	79	85
	C4.5	94	3	96	96
	MLP	94	2	98	97
webchess-html	NB	46	24	46	67
	C4.5	77	8	81	88
	MLP	73	8	80	86
utopia-html	NB	77	25	48	76
	C4.5	71	5	80	89
	MLP	94	2	94	97
yapig-html	NB	17	27	20	57
	C4.5	50	13	60	76
	MLP	67	27	50	71
myadmin-html	NB	75	19	60	79
	C4.5	94	0	100	98
	MLP	94	2	94	97
Average	NB	67	31	57	72
	C4.5	78	13	82	86
	MLP	84	15	81	86

(a)

Measure (%)		<i>Pd</i>	<i>Pf</i>	<i>Pr</i>	<i>Acc</i>
Data & Classifier					
geccbblite-sql	NB	75	20	75	78
	C4.5	75	20	75	78
	MLP	100	20	80	89
schoolmate-sql	NB	84	38	90	79
	C4.5	93	27	93	89
	MLP	95	27	94	91
faqforge-sql	NB	88	8	88	91
	C4.5	100	8	90	95
	MLP	94	8	89	93
webchess-sql	NB	88	21	78	83
	C4.5	100	14	86	93
	MLP	92	17	82	87
Average	NB	82	22	84	83
	C4.5	90	18	86	88
	MLP	97	18	88	91

(b)

Figure 3-3. Classification results of (a) XSS vulnerability predictors (b) SQLI vulnerability predictors, built from static code attributes.

As shown in Figure 3-3, the MLP and C4.5 models achieved $acc \geq 85\%$ and $pr \geq 80\%$. These results can be interpreted as at least 8 out of 10 cases are correctly predicted and at least 8 out of 10 vulnerable cases reported by our predictors are worth investigating for security audits. Although we did not compare the results directly, in general, these results are better than the results reported by static-analysis-based vulnerability detection approaches such as [61, 82, 165], which are known for low precision [68]. For example, Xie and Aiken's experiments [165] reported

15 false warnings for database configuration variables and information derived from database queries and configuration settings, which are actually benign. *Pixy* [61] also reported 47 false alarms (~50% of total reports) due to the use of regular expressions, “if”-constructs, and file includes. Since we re-used some of the test subjects used in their experiments, we observed that most of the above cases were appropriately handled by our predictors. This is because our predictors learned from those coding patterns causing vulnerabilities and applied the knowledge in predicting new instances. The advantage of data miners is that if there are sufficient sample data containing consistent information, they can be easily updated (i.e., re-trained).

However, our predictors are not without flaws. In the experiments, we encountered cases that they may not appropriately handle. We observed that regular expressions are commonly used in web applications for both input validation and sanitization purposes. But as observed by Jovanovic et al. [61] and Xie and Aiken [165], these expressions are often used incorrectly. For example, see a case from Utopia News Pro\users.php (reported by Xie and Aiken [165]):

```

1 if( !ereg( '[0-9]+', $userid ) ) {
2     unpm_msgBox( 'Invalid user ID!' );
3     exit;
4 }
5 $getuser = $DB->query( "SELECT * FROM `unp_user` WHERE userid='$userid'" );

```

In the above case, the regular expression “[0-9]+” was erroneously used to validate `$userid` as a number. The expression in fact only checks for the existence of a number. Our predictors work well as long as developers consistently make similar mistakes (because, assuming other attributes are irrelevant, the predictor would classify an instance as vulnerable if there is regular-expression-based-matching). But when such mistakes become rare, there is a danger of high false alarm rates. This is because, as predictors are trained on past data, they inherently struggle when the instance to be predicted is represented with inconsistent data. Intuitively, this problem can be alleviated by more precise classification methods. For example, we could refine the abstracted classification of regular-expression-based matching (*Regex-match*) with more in-depth classifications such as regular-expression-based all-numeric-check, regular-expression-based HTML-tags-check, etc. However, this task shall require complex program analysis techniques.

To perform data reduction, in *Weka*, for each dataset, we ran the gain ratio attribute evaluator. The frequencies of the top eight attributes ranked by the evaluator are shown in Figure 3-4. As experienced by Menzies et al. [91], different datasets selected different sets of best 8 attributes, except for some repeatedly selected attributes such as *SQL*, *Text-database*, and *Propagate*. That explains the total of 19 (out of available 27) “best” attributes selected by 11 datasets.

Using these best attributes, we investigated whether there are any redundant attributes that we could omit to improve a classifier’s performance. We re-ran the above 5-fold cross validation process on the same datasets in *Weka* but using only the best 3, 5, or 8 attributes. Then, we computed the *balance* results of each classifier. The results are plotted in Figure 3-5. Again, we used Wilcoxon signed-rank test to report if a classifier built with best 3, 5, or 8 attributes is statistically better than the same classifier built with all attributes.

Attribute	Frequency	
	#	%
SQL	9	82
Text-database	9	82
Propagate	8	73
Uninit	7	64
Null	7	64
Size	7	64
Database	6	55
Match	6	55
Un-taint	6	55
Other	6	55
Session	4	36
HTML	4	36
Containment	2	18
Custom	2	18
Client	1	9
Regex-match	1	9
XSS-sanitization	1	9
Encryption	1	9
Replacement	1	9

Figure 3-4. Best attributes selected by *Weka*’s gain ratio ranker for the 11 datasets shown in Table 3-3.

Figure 3-5. Balance results over 11 datasets of each classifier learned on best 3, 5, 8, or all attributes.

In Figure 3-5a, there are noticeable upward shifts in the lower and/or upper quartiles for best 3, 5, or 8 attributes compared to all attributes. That is, the NB model achieved more performance increases over the 11 datasets when built with best 3, 5, or 8 attributes instead of all attributes. Information provided by some attributes might have hurt the NB model built with all attributes. This result, however, validates the findings by Hall and Holmes [51] and Lessmann et al. [75] that data preprocessing activities such as feature selection could improve simple classifiers like NB classifier.

Even so, our Wilcoxon tests reveal that the improvements of the NB model were statistically insignificant. Similarly, although there are more performance increases over the 11 datasets for the MLP model built with best 5 attributes (Figure 3-5c), improvements were statistically insignificant. In fact, the C4.5 and MLP models' performances decreased more over the 11 datasets when built with best 3 or 8 attributes (Figure 3-5b and Figure 3-5c). Performance degradation for the MLP model built with best 8 attributes (compared to the MLP model built with all attributes) was statistically significant at 95%.

In summary, from these results, we can conclude that:

- The proposed static code attributes are useful indicators of web application vulnerabilities, supporting our hypothesis *H1*.
- Even though some attributes might be irrelevant or redundant for some classifiers or for different datasets, in general, it is better to use all the proposed attributes.

3.5.2 Threats to Validity

Our experiments, like every data mining experiments, may suffer from data sampling bias. Our predictors were trained using data from a set of applications which are all vulnerable to multiple SQLI or XSS vulnerabilities. However, all applications except *GeccBBLite* are real-world projects developed by professionals rather than novices or students. And these applications have been tested and verified for required functionalities before their releases. As such, we believe that our results are applicable to real-world applications.

Our classification schemes may not always be adequate for all cases. For example, functions such as `similar_text` are classified as *Un-taint* based on our observations of many web applications; however, it is possible that some sophisticated attackers may make use of the information returned from such functions to generate successful security attacks. However, the advantage of using data miners is that when such cases become significant, they can be easily re-trained with newly reported vulnerability information to become aware of similar cases in future.

Data sampling procedure might also affect the generalization of the experiment results. We used cross-validation method which is a well-established approach for data mining experiments. *Weka* also provides split-sample set up which has been used by some other prediction studies [7, 75, 149, 170]. We ran our experiments with this setup as well. However, since the differences between the results are minimal, we resorted to using only one data sampling method.

We used normalization technique to preprocess our data. But, there are many other data preprocessing methods such as logarithmic data filtering which might produce different results. Likewise, there are various attribute selection methods investigated in literature [41, 51]. We used information gain ratio technique. But, a different method might select different subset of attributes and alter the results. The choice of classifiers might also affect our empirical study because classification algorithms we used may over-fit or under-fit the datasets. We have used 3 classifiers with different learning models to avoid this bias. Interested researchers may try data mining methods different from our current setup or use more data mining activities in order to validate or improve our results.

For all the above threats, the best way to prove or refute our claims and results is to replicate our experiments. Researchers could do so since we have clearly defined our experiment methods and we also provide the datasets used in the experiment and the tool used to collect the data in our website [116].

3.6 Conclusion

In this chapter, we classified the types of inputs and sinks that may cause SQLI and XSS attacks. The types of input validation and sanitization methods that are commonly applied to inputs to avoid security issues are also classified. Consequently, static code attributes that reflect these classification schemes are proposed. For each sensitive sink in a web program, the static attribute vectors are extracted from the backward static program slice of the sink. Vulnerability prediction models are then built from the extracted information and the available vulnerability information.

We conducted experiments on a set of PHP-based web applications to investigate our hypothesis that the proposed static code attributes are important indicators of SQLI and XSS vulnerabilities. In the experiments, our best prediction model predicted SQLI and XSS vulnerabilities with the accuracy of $pd \geq 84\%$ and $pf \leq 15\%$ and thus, the results supported our hypothesis.

We do not claim that vulnerability prediction method is the replacement of existing vulnerability detection approaches because, in theory, predictors could only provide probabilistic remarks based on past data. But since our static attributes can be easily collected by simple static analysis techniques, the proposed method offers an alternative and cheap way of mitigating security vulnerabilities in web applications.

Chapter 4

SUPERVISED AND UN-SUPERVISED VULNERABILITY PREDICTION FROM HYBRID CODE ATTRIBUTES⁵

In Chapter 3, we proposed a set of static code attributes that characterize input validation and input sanitization code patterns. And we showed that some of the proposed static attributes are significant predictors of web application vulnerabilities related to SQL injection and cross site scripting. However, we still have works left to do to further improve or enhance the usefulness of our predictors. There are *two main problems* associated with the static analysis-based predictors presented in Chapter 3—(1) as the analysis being static, the predictors might be limited in terms of the prediction accuracy they can yield (the predictive capability of these attributes is dependent on the precision of the classifications of input validation and sanitization code patterns); (2) being a supervised learning based approach, its effectiveness is dependent on the availability of sufficient training data labeled with manually checked security vulnerabilities.

Static attributes have the advantage of reflecting general properties of a program. Yet, dynamic attributes collected from execution traces may reflect more specific code characteristics that are complementary to static attributes. Therefore, *dynamic analysis* can help improve the accuracy of our current static predictors. But, this would come at the expense of an additional dynamic analysis framework, which is generally difficult to adopt. Nevertheless, dynamic analysis would address our first problem above and would be useful for those who prioritize prediction accuracy.

To address the second problem, we incorporate an un-supervised learning method, *cluster analysis*, into our vulnerability prediction framework and present techniques that achieve acceptable prediction accuracy so that the predictors are still useful in the absence of labeled training data. In existing vulnerability prediction studies, supervised learning methods are

⁵ © 2013 IEEE. Reprinted, with permission, from 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 35th International Conference on Software Engineering*, May 2013.

generally used. We have no knowledge of vulnerability prediction models built using unsupervised learning methods.

Hence, in this chapter, we address the above two limitations and provide a more extensive empirical study than that of our work presented in Chapter 3. On top of our original static analysis framework, we introduce a pattern mining approach based on dynamic analysis that classifies input validation and sanitization functions through the systematic execution of these functions and the analysis of their execution traces. We also use both supervised learning methods and unsupervised learning methods to build vulnerability predictors and determine the effectiveness of the predictors with and without labeled training data, respectively.

Contributions and Results

- We propose a set of hybrid attributes based on static analysis and dynamic analysis, characterizing common input sanitization and input validation code patterns.
- We modify our prototype tool *PhpMiner* presented in Chapter 3 to include dynamic analysis functionalities so that it extracts the proposed hybrid (static and dynamic) attributes from PHP programs.
- We build prediction models from hybrid attributes using both classification and cluster analysis techniques in order to predict vulnerabilities, in the presence or absence of labeled training data, respectively.
- In our experiments across six PHP applications, our new supervised vulnerability predictors based on hybrid attributes achieved, on average, 90% recall and 85% precision, that is a sharp increase in recall when compared to static analysis-based predictions. Though not nearly as accurate, our unsupervised predictors based on clustering achieved, on average, 76% recall and 39% precision, thus suggesting they can be useful in the absence of labeled training data.

This chapter is organized as follows. Section 4.1 proposes the hybrid code attributes. Section 4.2 defines the two hypotheses to be investigated in this chapter. Section 4.3 presents our vulnerability prediction framework. Section 4.4 explains the workings of our prediction approach with an example. Section 4.5 describes our prototype tool and the datasets collected for experiments. Section 4.6 evaluates our proposed vulnerability prediction framework. Section 4.7 concludes this chapter.

4.1 Hybrid Code Attributes

From our observations of many web applications, good security defense schemes are typically functions implemented independently from the main function of the program. And such functions typically consist of a sequence of string functions and operations designed to cleanse the inputs. In Chapter 3, we statically analyzed each individual program statement and extracted properties related to security in an isolated manner. The analysis was basically statement-based. In this chapter, we aim to broaden the analysis to both statement-based and function-based because it would allow predictors to learn of the good and the bad customized security functions implemented or adopted, thereby enhancing the prediction capability. In the following, we present the analysis method and propose the hybrid attributes.

Data dependence graph: Our unit of measurement is a sink. A sink is a node in a control flow graph of a web program that may cause SQLI or XSS attacks. Basically, a sink represents a program statement that interacts with a database (denoted as *SQL* sink) or web client (denoted as *HTML* sink). Given a sink k , we compute its data dependence graph (DDG_k) using data flow analysis. The graph provides reachable definitions for the variables used in the sink, that is, it contains the nodes on which the sink is data dependent [34]. As such, any input validation and sanitization operations implemented for the sink k can be found in the nodes in DDG_k .

In Chapter 3, we used backward static program slice to measure the properties of the sink related to security. But, in our experiments presented in Chapter 3, we observed that there are noises affecting the accuracy of the predictors (we found many nodes unrelated to vulnerability in program slices). This is mainly because program slice of a sink includes all nodes (all control dependent and data dependent nodes) that influence the sink [162]. For example, see the code snippet shown in the following.

```
1  $errmsg = $_GET['err_msg'];
2  $id = htmlspecialchars($_GET['id']);
3  if($id == null)
4      $error = 1;
5  if ($error == 1)
6      echo $errmsg;
```

The slice of *HTML* sink 6 consists of all the statements above in which statement 2 invokes a language-provided XSS-sanitization method. This might confuse the predictor in predicting the vulnerability of the sink because, regardless of the presence of an XSS sanitization function in the slice, the sink is still vulnerable. On the other hand, the data dependence graph of sink 6 only

consists of statement 1 and statement 6 and hence, noisy statements such as statements 2-5 are not included. But, this scenario could also be reversed as there is a possibility that such statements are important indicators of vulnerability instead of noises. Hence, we do not conclude that using data dependence graph is definitely better than using program slice. But, empirically, we achieved better results using data dependence graph in our overall experiments (on datasets used in both Chapter 3 and Chapter 4). Therefore, we preferred the use of data dependence graph in this chapter.

Hence, instead of nodes in a slice, here, we classify the nodes in DDG_k according to their security-related properties, and then we capture these classifications in a set of static-dynamic hybrid code attributes on which vulnerability predictors are to be built. Most of the static code attributes presented and used in Chapter 3 are re-used in this chapter, but some of those attributes are omitted because we now classify some functions and operations, such as string replacement, using dynamic analysis for better accuracy.

To classify nodes in DDG_k , we use a hybrid approach that combines static analysis and dynamic analysis techniques. From the language built-in functions that have specific security purposes (e.g., `addslashes`), the language operators (e.g., string concatenation operator “.”), or the predefined language parameters (e.g., `$_GET`) used in a given node n in DDG_k , n is classified *statically*. But it is classified *dynamically* if it invokes user-defined functions or some built-in functions such as string replacement and string matching functions. As a control flow node n may contain a variety of program operations, there may be multiple classifications for n . We shall address the attributes on which the classification schemes will rely as *hybrid attributes*. The attributes are listed in Table 4-1 and presented next.

Static analysis-based classification: Some of the language built-in functions and operations can be precisely classified from their static properties or specific purposes. The classification can be carried out by simply checking the properties of the function or operation. Attributes 1-15 in Table 4-1 characterize the functions and operators to be classified statically. As these attributes are the same as those presented and used in Chapter 3, we shall only briefly review them.

Table 4-1. Static-dynamic hybrid attributes

Attribute ID	Attribute Name	Description	Data Type
Static analysis attributes			
1	Client	The number of nodes that access data from HTTP request parameters	Numeric
2	File	The number of nodes that access data from files	Numeric
3	Database	The number of nodes that access data from database	Numeric
4	Text-database	Boolean value 'TRUE' if there is any text-based data accessed from database; 'FALSE' otherwise	Boolean
5	Other-database	Boolean value 'TRUE' if there is any data except text-based data accessed from database; 'FALSE' otherwise	Boolean
6	Session	The number of nodes that access data from persistent data objects	Numeric
7	Uninit	The number of nodes that reference un-initialized program variable	Numeric
8	SQLi-sanitization	The number of nodes that apply standard sanitization functions for preventing SQLi issues	Numeric
9	XSS-sanitization	The number of nodes that apply standard sanitization functions for preventing XSS issues	Numeric
10	Numeric-casting	The number of nodes that type cast data into a numeric type data	Numeric
11	Numeric-type-check	The number of nodes that perform numeric data type check	Numeric
12	Encoding	The number of nodes that encode data into a certain format	Numeric
13	Un-taint	The number of nodes that return predefined information or information not influenced by external users	Numeric
14	Boolean	The number of nodes which invoke functions that return Boolean value	Numeric
15	Propagate	The number of nodes that propagate the tainted-ness of an input string	Numeric
Dynamic analysis attributes			
16	Numeric	The number of nodes which invoke functions that return only numeric, mathematic, or dash characters	Numeric
17	LimitLength	The number of nodes that invoke string-length limiting functions	Numeric
18	URL	The number of nodes that invoke path-filtering functions	Numeric
19	EventHandler	The number of nodes that invoke event handler filtering functions	Numeric
20	HTMLTag	The number of nodes that invoke HTML tag filtering functions	Numeric
21	Delimiter	The number of nodes that invoke delimiter filtering functions	Numeric
22	AlternateEncode	The number of nodes that invoke alternate character encoding filtering functions	Numeric
Target attribute			
23	Vulnerable?	Indicates a class label—Vulnerable or Non-vulnerable	Boolean

Depending on the nature of sources, we categorize the inputs into seven types as explained by attributes 1-7 in Table 4-1. Attributes 8-13 basically involve language built-in functions and

operators that could be used in input validation and sanitization procedures. Attribute 8 and 9 correspond to language-provided SQLI and XSS sanitization routines respectively. Attribute 10 involves type casting built-in functions or operations (e.g., `$a = (double) $b/$c`) that cast the input string into a numeric type data. Attribute 11 corresponds to language-provided numeric data type checking functions (e.g., `is_numeric`). Attribute 12 corresponds to encoding functions. An input variable may be properly sanitized using encoding functions (e.g., `<a href='login.php?name='.urlencode(input)>`). Attribute 13 matches to functions or operations that return predefined information or information not extracted from the input string (e.g., `mysql_num_rows`). The attribute *Boolean* is a new static code attribute we introduce in this chapter. It is included as a type of validation and sanitization because a Boolean value returned from a (user-defined or built-in) function is definitely safe for use in a sink. And such a function can be classified statically by checking its function protocol.

Clearly, nodes in DDG_k may also include ordinary operations that do not serve any security purpose and that just propagate the tainted data. Therefore, functions and operations that are not classified as any of the rest of types via either static analysis or dynamic analysis (discussed in the following) are classified into the attribute *Propagate*.

Dynamic analysis-based classification: When a node invokes a user-defined function or a language built-in string replacement/matching function (such as `str_replace`), the type or purpose of the function cannot be easily inferred from static analysis. Since inputs to web applications are naturally strings, string replacement/matching functions are generally used to implement input validation and sanitization procedures. A good security function generally consists of a set of string functions that allow only valid strings or filter unsafe strings. A filtering action entails character removal or escaping.

In Chapter 3, we simply characterized such string functions with attributes such as *Match* (e.g., `strcmp`) and *Regex-replacement* (e.g., `preg_replace`). This is too general and our static code attributes could not discriminate correct and incorrect string functions (e.g., it treats regular expression-based string replacement functions as either correct or incorrect). Hence, to improve the accuracy of classification, in this chapter, dynamic analysis is used if a node in DDG_k invokes a user-defined function or a language built-in string replacement/matching function. The dynamic analysis attributes are defined as follows:

- 1) *Numeric*: functions that return only numeric, mathematic, and/or dash '-' characters (e.g., functions that validate inputs such as mathematic equations, postal code, or credit card number).

- 2) *LimitLength*: functions that limit the length of an input string to a specified number.
- 3) *URL*: functions that filter directory paths or URLs (e.g., `<a href src='www.hack.com/hack.js'`).
- 4) *EventHandler*: functions that filter event handlers such as `onload`.
- 5) *HTMLTag*: functions that filter HTML tags (e.g., strings between `<` and the first white space or `>`).
- 6) *Delimiter*: functions that filter delimiters which could disrupt the syntax of intended HTML documents or SQL queries (e.g., string-delimiters such as single quote and double quote; comment-delimiters such as `/*`, `#`, `//`, and `--`; and some other special characters such as parenthesis, semi-colon, backslash, null byte, and new line).
- 7) *AlternateEncode*: functions that filter alternate character encodings (e.g., `char(0x27)`).

Note that though the dynamic attribute *Numeric* is similar to static analysis attributes 10 and 11 (Table 4-1), those two attributes characterize the nodes that invoke language-built-in-specific numeric type casting operations and numeric type checking functions, respectively.

We believe that the above attributes reflect the types of input validation and sanitization methods that are commonly used to prevent SQLI or XSS attacks. Clearly, a user-defined function or a string replacement/matching function may correspond to more than one attribute. If a function corresponds to attributes *A* and *B*, then, both the values of *A* and *B* are to be incremented (if *A* and *B* are numeric attributes). In detail, (1) we maintain seven sets of test inputs derived from XSS and SQLI cheat sheets provided by OWASP [107, 109] and RSnake [118, 119]. These two security specialists provide a comprehensive coverage of XSS and SQLI attack vectors that could filter many types of input validation and sanitization routines. Each set of test inputs (denoted as *test-attr-set*) tests for each dynamic analysis attribute (e.g., a test input `<p>test</p>` tests for attribute *HTMLTag* as it could discriminate functions that accept or reject HTML tags); and (2) for a *test-attr-set* *T* that tests for an attribute *A*, we execute the concerned function with a test input t_1 from *T* and check if the function corresponds to *A* by analyzing the function output. If the function cannot be classified as *A*, we choose a different test input t_2 and repeat the process until it is classified as *A* or all the test inputs from *T* have been used; (3) step 2 is iterated for all the seven *test-attr-sets*, each set testing for each dynamic analysis attribute.

Not all function arguments are associated with user inputs. Some arguments are assigned with literal values in the program. Such *literal arguments* can be easily identified from the nodes in DDG_k . Test inputs are only assigned to arguments that are derived from user inputs and literal arguments are assigned with their own literal values extracted using data flow analysis. More than

one value is also possible for a literal argument if there are conditional branches. It is logical as the same function can be used to sanitize a variable differently depending on the program path along which the variable is propagated. For each possible value of a literal argument, we repeat the above dynamic classification process. As explained, we expect some functions to match multiple classifications.

Attributes 16-22 in Table 4-1 represent the dynamic analysis-based classifications presented above. We shall provide more details on the classification methods in our example section, Section 4.4.

Target attribute: The last attribute *Vulnerable?* is the target attribute which is used to indicate the class label to be predicted.

Attribute vector: The attribute vector representation is the same as that described in Chapter 3. Except, here we propose 23 hybrid attributes. Hence, every sink is characterized by a 23-dimensional attribute vector. The value of each attribute in that vector is either numeric or Boolean. As shown in Table 4-1, 20 attributes are numeric and 3 attributes are Boolean.

4.2 Research Hypotheses

In this section, we investigate the two research hypotheses presented in the following.

Our first goal is to build accurate supervised vulnerability prediction models from the hybrid attributes presented earlier. This leads to our first hypothesis (*H1*): Given a sufficient sample of vulnerability data, classifiers learned from such data are accurate at vulnerability predictions.

Hence, if *H1* is true, we would see high recalls and low false alarm rates from the prediction models built from hybrid attributes. We should also expect that the model built from these hybrid attributes generally performs better than the one built from the static attributes alone.

Our second goal is to build vulnerability prediction models that work in the absence of sufficient labeled data. Although classifiers can be effective, a sufficient number of instances with known vulnerability information is required to train a classifier. It is usually tedious and labor-intensive to tag many instances with vulnerability labels. Sometimes, the vulnerability information is not even yet known. In such situations, supervised training (i.e., where training instances need to be labeled with vulnerability information) is simply not feasible.

Cluster analysis, on the other hand, is a type of un-supervised learning methods in which no class label is required for training with instances. Intrusion detection studies such as Portnoy et al. [113] and Thamaraiselvi et al. [146] have shown that cluster analysis could identify numerous

anomalies (intrusions in their context) based on the two assumptions that (1) normal instances are much more frequent than anomalies and (2) anomalies have characteristics different from normal instances. If, in our context, the same two assumptions hold, cluster analysis could be used for identifying vulnerable sinks as well. This leads to our second hypothesis (*H2*): Vulnerable sinks can be distinguished from non-vulnerable sinks based on the hybrid attributes proposed above.

If *H2* is true, we would observe that cluster analysis on the unlabeled instances containing the data of hybrid attributes can predict vulnerabilities. Hence, when classification-based vulnerability prediction models are not a feasible option, our approach also includes making use of clustering for building vulnerability prediction models from the hybrid attributes when the above assumptions are met.

4.3 Vulnerability Prediction Framework

4.3.1 Data Preprocessing

Normalization: Like in Chapter 3, we use min-max normalization to standardize our data (we use the same scale of zero to one) and to avoid attribute biasing. Especially in the context of cluster analysis where similarity measurement combines multiple attribute scales, attributes with large-scale values could influence the predictor and affect its prediction accuracy.

Principal component analysis: Principal component analysis (PCA) is a useful technique to identify linearly uncorrelated dimensions in a large datasets with possibly many inter-correlated attributes. Multivariate data mining and statistical techniques used to build classifiers, such as logistic regression, see their performance negatively impacted in the presence of numerous inter-correlated attributes. PCA results in a new set of attributes (principal components), each of which is a linear combination of some of the original attributes. The number of principal components is usually much smaller.

4.3.2 Building Supervised Vulnerability Predictors

Classifiers: Classification is a type of supervised learning methods because the class label of each training instance has to be provided. In our experiments in Chapter 3, Multi-Layer Perceptron (MLP) performed better than the other two classifiers—Naïve Bayes and C4.5. Therefore, in this work, we shall re-use MLP and also try a very different classification algorithm called Logistic Regression (LR) in an attempt to further explore more data mining algorithms and optimize

accuracy. Hence, we shall build LR and MLP models for this experiment. These classifiers were ranked as among the top classifiers in recent studies [75].

We have discussed the workings of an MLP in Chapter 3. For the values of parameters used in MLP, we use the same settings used in Chapter 3, which are defaults set by *Weka*:

```
No. of hidden layer = 1;
No. of neurons per hidden layer = (no. of attributes + no. of classes)/2
```

Logistic Regression is a type of statistical regression models. It can be used to predict the outcome of a dependent variable (in our case, the target attribute *Vulnerable?*) based on a set of predictor variables (static and dynamic analysis attributes). The equations of Logistic Regression are given by Cessie and Houwelingen [22] as shown in the following:

If there are c classes for n instances with m attributes, the parameter matrix B to be calculated will be an $m \times (c-1)$ matrix.

The probability for class j with the exception of the last class is:

$$P_j(X_i) = \exp(X_i \cdot B_j) / ((\sum_{j=1..(c-1)} \exp(X_i \cdot B_j)) + 1)$$

The last class has probability:

$$1 - (\sum_{j=1..(c-1)} P_j(X_i)) = 1 / ((\sum_{j=1..(c-1)} \exp(X_i \cdot B_j)) + 1)$$

The (negative) multinomial log-likelihood is thus:

$$L = -\sum_{i=1..n} \{ \sum_{j=1..(c-1)} (Y_{ij} * \ln(P_j(X_i))) + (1 - (\sum_{j=1..(c-1)} Y_{ij})) * \ln(1 - \sum_{j=1..(c-1)} P_j(X_i)) \} + \text{ridge} * (B^2)$$

In order to find the matrix B for which L is minimized, a Quasi-Newton Method is used to search for the optimized values of the $m \times (c-1)$ variables.

To build a LR model using *Weka*, it is required to set the two parameters—*maxIts* and *ridge*. The parameter *maxIts* defines the maximum number of iterations to perform. The parameter *ridge* is an estimator used in the log-likelihood. Again, like the MLP model, we use the default values in *Weka* which sets *maxIts* to -1 meaning the iteration is continued until the attribute-class weights are fully optimized. The value of *ridge* is set as $(1 * \exp(-8))$.

As also discussed in Chapter 3, we do not fine tune the parameter values in an attempt to optimize any particular classifier since our focus is to rather endorse the predictive capability of our proposed attributes using different types of classifiers with default or commonly-used settings. Further details about these classification techniques can be found in Witten and Frank [163].

Training and testing: As we aim to conduct more comprehensive experiments, we chose the test subjects with relatively larger datasets than those used in Chapter 3. Hence, for data sampling,

we are able to use recommended [53, 163] and commonly-used [41, 90, 91, 138, 141] stratified 10-fold cross validation setup.

Performance Measures: To evaluate our supervised vulnerability predictors, we shall use the same measures used in Chapter 3. That is, we compute recall or probability of detection (pd), probability of false alarm (pf), and precision (pr) for every prediction output of the classifiers.

4.3.3 Building Un-Supervised Vulnerability Predictors

Cluster analysis: Unlike classification methods, cluster analysis works in the absence of class labels for training instances. But its predictive capability would be expected to be inherently lower due to the lack of supervision. Like Portnoy et al.'s un-supervised intrusion detection study [113], the performance of our cluster analysis here should depend on the following two assumptions: (1) non-vulnerable sinks are much more frequent than vulnerable sinks and (2) vulnerable sinks have characteristics different from non-vulnerable sinks. If these two assumptions are met and $H2$ is true, vulnerable sinks would be clustered together as outliers in terms of hybrid attribute values, which could then be detected by cluster analysis.

Since there is no need to label instances, un-supervised learning, such as cluster analysis, is expected to be much less expensive than building classifiers for vulnerability prediction.

We shall evaluate k -means clustering algorithm applied to our proposed hybrid attributes. K -means is a simple and often effective partitioning algorithm. Given an input k , it partitions a set of instances into k clusters in such a way that similarity among instances is maximized within the same clusters and minimized across the different clusters. For similarity measurement, standard distance functions can be used. The *Euclidean distance function* is selected for our experiments. Further details about the algorithm are provided in [163].

Parameter estimation: As clustering only groups instances based on their similarities, some parameters must be defined to label the clusters as 'Vulnerable' or 'Non-vulnerable'. The problem here is "*Given a set of clusters produced by a clustering algorithm, what are the best rules (parameters) to single out clusters that contain a large proportion of vulnerable sinks?*". In Portnoy et al.'s clustering-based intrusion detection study [113], a parameter N was introduced as the percentage of the largest clusters that would be labeled as 'normal cluster'. The value $N=15\%$ was used as it was found to optimize their results.

For our clustering-based vulnerability prediction study, we define a parameter called *%Normal*. It is defined as the minimum size (in terms of percentage of instances) of clusters that would be labeled as 'Non-Vulnerable'. For example, if *%Normal=10*, the clusters containing more than 10%

of data would be labeled as ‘Non-Vulnerable’. Since one assumption of our cluster analysis approach is that non-vulnerable instances outnumber vulnerable instances, we expect the value of *%Normal* to be large. As required by *k*-means algorithm, we also need to determine a parameter *k* that indicates the number of clusters to be produced by *k*-means. The ideal number of clusters is two, one being vulnerable cluster and the other non-vulnerable cluster. But, practically we expect the value of *k* to be more than two because non-vulnerable instances will not always be so similar and thus these instances may not be grouped into a single cluster. The same goes for vulnerable instances.

We determined the two parameters by performing experiments that optimize results on the test subjects used to evaluate our static analysis-based approach presented in Chapter 3. In the experiments, we observed that by setting *%Normal*=12, we can identify two vulnerable clusters and two non-vulnerable clusters for most datasets, and three non-vulnerable clusters and one vulnerable cluster for some datasets. Hence, we concluded the optimal parameter values as *k*=4 and *%Normal*=12. We use these two parameter values in building the clustering-based vulnerability prediction model. It is suggested that to apply our cluster analysis-based approach, like us, one also has to perform preliminary experiments on the available datasets to find the optimal parameter values that could achieve good prediction performances on future datasets.

Performance Measures: Again, we shall use the same performance measures—*pd*, *pf*, and *pr*, to evaluate this un-supervised prediction model.

4.4 Example

In this section, we explain in detail the classification methods and the attribute collection process using the program in Figure 4-1.

Statement 1 is a class of *input* because it accesses an HTTP session parameter. It can be statically classified via checking the accessed, predefined parameter `$_SESSION`. Statement 2 can be classified as *XSS-sanitization* because it invokes a standard escaping routine. Again, it can be statically classified via checking the function name; that is, we predefine the function `htmlspecialchars` as an XSS sanitization type. Statement 3 is an *Un-taint* type.

Figure 4-1b shows the data dependence graph of *HTML* sink 6 in Figure 4-1a. Node 4 invokes a user-defined function and it is clear that it could not be precisely classified by just looking up the predefined classifications. We classify such nodes via dynamic analysis as explained below.

In node 4, a customized security function `PMA_backquote` is invoked with two arguments `$trg_db` and `$sqlEscape`. By a data flow analysis, the literal value ```` assigned to `$sqlEscape` is extracted from node 3. From node 1, `$trg_db` is identified as an input variable. It is then assigned with a value obtained from *test-attr-sets* (see *Dynamic analysis-based classification* in Section 4.1). And the function is executed multiple times (each time selecting a different value from *test-attr-sets*) to determine if it can be classified as one or more dynamic analysis attributes. Classifications are carried out based on the types of input values used and the contents of the resulting outputs. When a test input such as ``1` or `1=1` is used, the returned result shall be ```1` or `1=1` and the function would be classified as *Delimiter* as it escapes a string-delimiter ```. Nodes 9 and 10 shall not be classified as the nodes are contained in the user-defined function that has already been classified.

```

1  $trg_db = $_SESSION['trg_db'];
2  echo '<table><tr><th>Target database: ' .
    htmlspecialchars($trg_db);    //HTML sink

3  $sqlEscape = ``;

4  $query = "UPDATE " . PMA_backquote($trg_db,
    $sqlEscape) . " SET ...";

5  if ($display == true) {
6      echo "<p>" . $query . "</p>"; //HTML sink
    }

7  $rs = mysql_query($query);    //SQL sink

function PMA_backquote($a_name, $replace) {
8      if (strlen($a_name) && $a_name !== '') {
9          return `` .
            str_replace(``, $replace, $a_name) . ``;
        } else {
10         return $a_name;
        }
    }

```

(a)

(b)

Figure 4-1. (a) Sample vulnerable PHP code extracted from PhpMyadmin\server_synchronize.php (slightly modified for illustration purpose). The code cleanses an input using standard and customized sanitization functions. (b) Data dependence graph of sink statement 6.

Based on the above classifications, the attribute vectors for *HTML* sinks 2 and 6 extracted from their respective data dependence graphs would be represented as shown in the following:

Attribute sinks	Session	XSS-sanitization	Un-taint	Delimiter	Propagate	...	Vulnerable?
2	1	1	0	0	1	...	No
6	1	0	1	1	1	...	Yes

In the above table, the “...” refers to the rest of the attributes listed in Table 4-1, not shown here due to space limit. Their values would be either numeric or Boolean. Every sink would be represented by a 23-dimensional attribute vector like the ones above.

If we apply principal component analysis to the above two *HTML* sinks, it would result in one principal component as shown in the following:

$$0.577\text{Delimiter}-0.577\text{XSS-sanitization}+0.577\text{Un-taint}$$

Classification and cluster analysis outputs can be explained as follow. Say, for example, if there are 10 vulnerable sinks with the same attribute vectors as the above *HTML* sink 6 and there are 90 non-vulnerable sinks with the same attribute vectors as the above *HTML* sink 2, stratified 10-fold cross validation on both LR and MLP models will produce the following confusion matrix:

```

Correctly Classified Instances      100          100      %
Incorrectly Classified Instances      0           0      %
      a  b  <-- classified as
      10  0  |  a = Vulnerable
      0  90 |  b = Non-Vulnerable

```

And performance measures computed from the above matrix would be 100% *pd*, 0% *pf*, and 100% *pr*.

k-means clustering with the parameters set as *k*=4 and %*Normal*=12 would result in the following output:

```

Clustered Instances
      0      90 ( 90%)
      1      10 ( 10%)

      0  1  <-- assigned to cluster
      0  10 | Vulnerable
      90  0 | Non-Vulnerable

Cluster 0 <-- Non-Vulnerable
Cluster 1 <-- Vulnerable

```

And also in this case, we will have the performance measures computed as 100% *pd*, 0% *pf*, and 100% *pr*.

4.5 Prototype Tool and Dataset

We enhanced the tool *PhpMiner*, which was presented in Chapter 3, to include dynamic analysis and adapt to the new approach. The tool is modified such that it computes data dependence graph for each sink (for auditing purposes, *PhpMiner* also produces data dependence graph outputs, which can be viewed via graphic tools such as Graphviz [46]) and collects hybrid attributes from each node in the graph. Figure 4-2 illustrates the architecture of our modified prototype tool.

Figure 4-2. Architecture of enhanced *PhpMiner*

As shown in Figure 4-2, we use *Pixy* to extract the CFG and from the CFG, inputs and sinks are identified. Data dependence graph extractor (*DDG Extractor*) module extracts data dependence graphs for each sink using data flow analysis APIs provided by *Pixy*. *Static Classifier* module first attempts to collect static attributes by extracting and analyzing static properties of the *DDG* nodes. *Dynamic Analyzer* module is activated only when a node includes user-defined functions or language built-in string replacement/matching functions, which could not be accurately classified

by *Static Analyzer*. No classification is made for nodes that are contained in dynamically classified user-defined functions to avoid unnecessary or overlapping classifications. To identify function arguments (i.e., literals or inputs), static data flow analysis is used. Test inputs are generated from our predefined test suite which reflects the dynamic classification scheme proposed in Section 4.1. Dynamic execution of PHP functions is performed in *Function Executor* module, which makes use of PHP-method invocation utilities provided by *Php-Java Bridge* module implemented using APIs from a PHP/Java Bridge Java package (from <http://php-java-bridge.sourceforge.net/pjb/>). Function outputs are then analyzed to infer the possible type of validation and sanitization scheme by *Dynamic Classifier* module.

Using modified *PhpMiner*, we collected attribute vectors from six real-world PHP-based web applications downloaded from SourceForge [143]. Table 4-2 shows relevant statistics for these test subjects. The sizes of some of the test subjects used in Chapter 3 are small and thus, the experiments conducted in Chapter 3 might not have been comprehensive. Hence, in this chapter, we re-used only some of the test subjects from Chapter 3 and we added larger-scale test subjects such as *Phorum5.2.18* and *PhpMyAdmin3.4.4*. As shown in Table 4-2, the sizes of our test subjects now range from 2k-44k LOC. Furthermore, almost all the datasets experimented in Chapter 3 contains high percentages (>30%) of vulnerabilities. Such data is generally easier to predict than the data with low vulnerability count. Models predicting on such data could achieve good results. Hence, in this chapter, we use a few datasets with low percentages of vulnerabilities (<10%) to conduct our experiments more comprehensively and show that our models are applicable for applications with high or low vulnerability counts.

The last column in Table 4-2 shows the security advisories, such as CVE [27], from which the test subjects' vulnerability information is obtained. Some of these test subjects have also been benchmarked for the evaluation of a few major vulnerability detection approaches [61, 68, 159, 165]. Table 4-3 shows the datasets collected by *PhpMiner*. It shows the statistics of the *SQL* sinks and *HTML* sinks collected. In total, we collected 10 datasets (only 4 sets of *SQL* sinks were used as we have not tagged the vulnerability labels for *SQL* sinks in *PhpMyAdmin* and *Utopia* systems). Column 3 in Table 4-3 shows the number and percentage of vulnerable sinks in each dataset (manually inspected and tagged by us). On our website [116], we provide the implementation details of *PhpMiner* and the datasets.

Table 4-2. Statistics of the test subjects

Test Subject	Description	LOC	Security Advisories
SchoolMate 1.5.4	A tool for school administration	8145	Vulnerability information in [65]
FaqForge 1.3.2	Document creation and management	2238	Bugtraq-43897
Utopia News Pro 1.1.4	News management system	5737	Bugtraq-15027
Phorum 5.2.18	Message board software	12324	CVE-2008-1486 CVE-2011-4561
CuteSITE 1.2.3	Content management framework	11441	CVE-2010-5024 CVE-2010-5025
PhpMyAdmin 3.4.4	MySQL database management	44628	PMASA-2011-14 – PMASA-2011-20

Table 4-3. Datasets

Dataset	#HTML sinks	#Vuln. sinks (%Vuln.)	Principal components
schmate-html	172	138 (80%)	7
faqforge-html	115	53 (46%)	7
utopia-html	86	17 (20%)	9
phorum-html	237	9 (4%)	9
cutesite-html	239	40 (17%)	10
myadmin-html	305	20 (7%)	9
Dataset	#SQL sinks	#Vuln. sinks (%Vuln.)	Principal components
schmate-sql	189	152 (80%)	7
faqforge-sql	42	17 (40%)	3
phorum-sql	122	5 (4%)	6
cutesite-sql	63	35 (56%)	7

4.6 Evaluation

To investigate the two research hypotheses presented in Section 4.2, first we need to evaluate if our proposed hybrid code attributes can be used to build accurate vulnerability predictors. Then we need to evaluate if vulnerable sinks can be distinguished from non-vulnerable sinks based on the hybrid attributes.

Therefore, for evaluation, as described in Section 4.3, we first applied min-max normalization and then PCA to every dataset collected (Table 4-3). We used a subset of principal components as attributes such that the selected explains at least 95% of the data variance. The last column in Table 4-3 shows the numbers of principal components selected for building supervised and unsupervised vulnerability predictors.

4.6.1 Result on Supervised Vulnerability Prediction

The two selected classifiers are implemented in *Weka* [163]. We configured the desired classifier and the data sampling mode (10-fold cross validation) in *Weka*, provided the dataset, and ran *Weka* on a Pentium 3.4GHz 4GBRAM Windows XP machine. Each classifier was run on each dataset. Figure 4-3 shows the results.

As shown in Figure 4-3, on average, both MLP and LR models showed good performances with high vulnerability detection rates ($\geq 74\%$) and low false alarm rates ($\leq 8\%$). But on some datasets such as *phorum-html* and *phorum-sql*, the MLP model could not discriminate vulnerabilities whereas the LR model was able to. Therefore, based on current results, we advise to the use of LR to build vulnerability prediction models.

The significantly low false alarm rates achieved by our new models indicate that the models' accuracy has improved from static analysis-based models presented in Chapter 3. Yet, to provide an exact comparison baseline, we also built LR models from static analysis attributes alone and evaluated them in the same way as the above models. Results are shown in Figure 4-4. On average, our proposed LR models built from hybrid attributes achieved ($pd=16\%$, $pf=3\%$, $pr=2\%$) *improvements* over the LR models built from static analysis attributes only. As suggested by Demšar [30], we also used one-tailed Wilcoxon signed-ranks tests to perform pair-wise comparisons of the measures achieved by the two types of LR models over the 10 datasets. The tests show that the improvements of pd , pf , and pr were statistically significant at a 95% level, though only the increase in recall might be interesting from a practical standpoint.

We can conclude that dynamic analysis attributes contribute to significantly improving the accuracy of vulnerability predictors. Hence, these results support *H1*.

Measure (%)		<i>Pd</i>	<i>Pf</i>	<i>Pr</i>
Data & Classifier				
schmate-html	LR	99	3	98
	MLP	99	0	100
faqforge-html	LR	89	5	94
	MLP	91	5	94
utopia-html	LR	94	1	94
	MLP	94	2	89
phorum-html	LR	78	1	70
	MLP	33	0	100
cutesite-html	LR	68	9	61
	MLP	78	8	67
myadmin-html	LR	85	1	89
	MLP	75	1	83
Average results on XSS prediction	LR	86	3	84
	MLP	78	3	89
schmate-sql	LR	97	8	98
	MLP	96	35	92
faqforge-sql	LR	88	4	94
	MLP	88	4	94
phorum-sql	LR	100	3	63
	MLP	0	1	0
cutesite-sql	LR	91	14	89
	MLP	89	18	86
Average results on SQLIV prediction	LR	94	7	86
	MLP	68	15	68
Overall average	LR	90	5	85
	MLP	74	8	81

Figure 4-3. Classification results of XSS and SQLI vulnerability predictors built from hybrid code attributes

Measure (%)		<i>Pd</i>	<i>Pf</i>	<i>Pr</i>
Data & Classifier				
schmate-html	LR	99	9	98
faqforge-html	LR	91	6	92
utopia-html	LR	88	3	88
phorum-html	LR	44	1	67
cutesite-html	LR	35	6	54
myadmin-html	LR	80	1	89
schmate-sql	LR	93	30	93
faqforge-sql	LR	88	4	94
phorum-sql	LR	40	1	67
cutesite-sql	LR	86	18	86
Overall average	LR	74	8	83

Figure 4-4. Classification results of XSS and SQLI vulnerability predictors built from static code attributes

4.6.2 Result on Un-Supervised Vulnerability Prediction

Regarding *H2*, we evaluated a clustering model learned from our proposed hybrid attributes.

We applied *k*-means cluster analysis with Euclidean distance function (implemented in *Weka* [163]) on the datasets. For identifying vulnerable clusters, the two parameters—*k*=4 and %*Normal*=12 (see Section 4.3.3), were consistently used for all the datasets. Since there is no need to label instances, un-supervised learning like cluster analysis is much less expensive than supervised learning like classification. But, we should also expect much less accuracy from such a model.

As discussed in Section 4.3.3, the working of our clustering model is based on the following two assumptions: (1) non-vulnerable sinks are much more frequent than vulnerable sinks and (2) vulnerable sinks have characteristics different from non-vulnerable sinks.

More than 40% of sinks in *schmate-html*, *faqforge-html*, *schmate-sql*, *faqforge-sql*, and *cutesite-sql* are vulnerable sinks (see %Vuln. in Table 4-3). These datasets clearly violate the first assumption above as they contain high vulnerability counts. We expect low predictive power from our clustering models for such datasets. Consequently, we separated the datasets which meet our assumptions from the ones that violate the assumptions, and performed separate evaluations. Results on the former datasets are shown in Figure 4-5a and results on the latter sets are shown in Figure 4-5b.

As shown in Figure 4-5a, the *k*-mean's detection rate is very good, especially on *utopia-html* and *phorum-sql* datasets. But its average precision is half that of the supervised models above. This is directly caused by the inherent weakness of the un-supervised learning scheme. It is also affected by trade-offs between detection rates and false alarms. The trade-offs mainly result from the parameter %*Normal*. With a high value of %*Normal*, we label more clusters as 'Vulnerable', thus possibly reducing precision. Tuning such a parameter must be done in context based on available resources for vulnerability detections.

As expected, as shown in Figure 4-5b, cluster analyses on datasets which violate our first assumption result in very low detection rates because many or all of the vulnerable sinks did not appear as outliers (in terms of hybrid attribute values) to our clustering model. *Pr* was also undefined for some datasets as both *pd* and *pf* were null.

From the results in Figure 4-5a, we can conclude that, if certain assumptions are met, cluster analysis on unlabeled instances using hybrid attributes can help accurately predict vulnerabilities, thus supporting *H2*.

Data \ Measure (%)	<i>Pd</i>	<i>Pf</i>	<i>Pr</i>
utopia-html	100	13	65
phorum-html	56	11	16
cutesite-html	70	20	41
myadmin-html	55	8	33
phorum-sql	100	7	38
Average	76	12	39

(a)

Data \ Measure (%)	<i>Pd</i>	<i>Pf</i>	<i>Pr</i>
schmate-html	9	0	100
faqforge-html	26	0	100
schmate-sql	3	32	29
faqforge-sql	0	0	undefined
cutesite-sql	0	0	undefined
Average	8	6	undefined

(b)

Figure 4-5. *k*-means cluster analysis results on the datasets (a) which meet the assumptions (b) which violate the assumptions

4.6.3 Threats to Validity

Our data only reflects known vulnerabilities that are reported in vulnerability databases. Hence, our vulnerability predictions based on classifiers do not account for undiscovered vulnerabilities.

The application of cluster analysis is limited by the two assumptions stated above. In our experiments, clustering-based prediction models could accurately isolate vulnerabilities in the datasets which satisfy those assumptions. However, it is unclear how frequently these assumptions hold in practice across systems and types of vulnerabilities. Furthermore, we estimated two parameters (*k* and %*Normal*) driving the accuracy of cluster analysis based on our experience with preliminary experiments. We used the same two parameters for all the datasets. The parameters worked well for our context but may not generalize well elsewhere. But as most of our test subjects such as *PhpMyAdmin* are widely-used, real-world applications, we believe that the above threats do not significantly affect our results although tuning the parameters may be required for some applications.

The use of different or more data preprocessing activities may also alter our results. For example, during our preliminary experiments, we tested the datasets with and without PCA (see Section 4.3.1). Results without PCA were significantly inferior to results with PCA for the majority of datasets though no significant differences were observed for some.

Different classification and clustering algorithms could result in different results. In our experiments, we used two very different classification algorithms which are statistical-based and network-based, respectively. We also tried other classifiers like C4.5 and Naïve Bayes, but the average results were similar. We have not tried another algorithm for clustering-based prediction, but we expect similar results if similar parameters (i.e., k and *%Normal*) are used.

Like all other empirical studies, our results are limited to the applied data mining processes, the test subjects, and the experimental setup used. One good solution to refute, prove, or improve our results is to replicate the experiments with new test subjects and probably with further data mining strategies. This can be easily done since we have clearly defined our methods and setup, and we also provide the data used in the experiments and the data collection tool on our website [116].

4.7 Conclusion

The goal of our work in vulnerability prediction is to aid security auditing and testing by providing probabilistic alerts about potentially vulnerable code statements. We propose attributes, based on hybrid static and dynamic code analysis, which characterize input validation and sanitization code patterns for predicting vulnerabilities related to SQLI and XSS. Given a security-sensitive program statement, we collect the hybrid attributes by classifying the nodes from its data dependency graph. Static analysis is used to classify nodes that have unambiguous security-related purposes. Dynamic analysis is used to classify nodes that invoke user-defined or language built-in string replacement/matching functions since classification of such nodes by static analysis could be imprecise.

We evaluated if these hybrid attributes can be used to build effective vulnerability predictors, using both supervised and un-supervised learning methods. The latter have, in practice, the advantage of not requiring labeled training data (with known vulnerabilities) but may be significantly less accurate. In the experiments on six PHP web applications, we first showed that the hybrid attributes can accurately predict vulnerabilities (90% recall and 85% precision on average for logistic regression). We also observed that dynamic analysis helped achieve much better accuracy than static analysis alone, thus justifying its application. Last but not least, when meeting certain assumptions, cluster analysis showed to be a reasonably accurate, un-supervised learning method when no labeled data is available for training (76% recall and 39% precision on average). But since it is not nearly as accurate as supervised learning, it should be considered as a trade-off between data collection cost and accuracy.

Chapter 5

VULNERABILITY AUDITING⁶

We have discussed in Chapter 3 that developers adopt or implement defensive coding methods, such as input validation and input sanitization, to guard against SQLI and XSS. Input validation checks the user input against required properties such that only inputs satisfying the required properties are accepted for further processing. If the required properties are adequately defined in a way that any input which could result in unintended query string or HTML output is rejected, then SQLI and XSS vulnerabilities can be avoided. However, it is often the case that input validation alone is insufficient to prevent security violations. Therefore, input sanitization is used as an alternative or additional method for the defense in depth against SQLI and XSS. Input sanitization escapes or filters away the potentially malicious characters such as ‘’.

However, sanitization mechanisms could also fail in protecting the applications because it is hard to completely sanitize all malicious characters. Moreover, if the server programs do not properly specify a character set to be used, more sophisticated hackers could use a variety of character encoding schemes to craft scripts that could circumvent most sanitization schemes used by web applications. For example, using Base64 encoding scheme, a malicious string ‘ OR 1 = 1’ can be replaced with `IicgTlIgMSA9IDEi` without using the commonly known bad character ‘’.

In short, defensive coding practices could ensure the absence of vulnerabilities in program source code. However, developers often face difficulty in getting these practices right [28, 50, 158].

On the other hand, existing vulnerability mitigating approaches including our vulnerability prediction approach (Chapter 3 and Chapter 4) mainly focus on identifying potential vulnerabilities in program source code or preventing potential attacks at runtime. These approaches do not emphasize systematic reporting of defense artifacts implemented in programs.

⁶ The majority of this chapter is from a postprint of a paper submitted to and accepted for publication in IET Software and is subject to Institution of Engineering and Technology Copyright. The copy of record is available at IET Digital Library.

Thus, the information provided by these approaches is limited. As developers, security auditors, or project managers would prefer that their applications are free from security risks, from time to time, they would conduct security audits on the adequacy of defense artifacts employed in web programs. But, it would be difficult for them to examine the adequacy of defense features from the limited information provided by these existing approaches. They may have to walk through the whole chunk of program source code. Therefore, there is a need for an assisted approach that systematically extracts the defense features implemented in code and aids the debugging of extracted features.

In light of this, in this chapter, we propose a code auditing approach for verifying the adequacy of defense artifacts implemented in programs in defending against SQLI and XSS. Based on empirical studies, we first define the possible code patterns for implementing SQLI and XSS defenses. We then use static control and data flow analysis to automatically extract all such defenses from source code. We also introduce a variant of control flow graph called *tainted-information flow graph* to model the extracted information. Based on the model, we provide guidelines to classify the outputs into one of the following cases: vulnerable, non-vulnerable, probably-vulnerable, or probably-non-vulnerable.

Our proposed method is neither a replacement nor an alternative solution to existing vulnerability mitigating methods, including our vulnerability prediction method proposed in Chapter 3 and Chapter 4. Rather, it aims to complement those methods by aiding the process of vulnerability auditing and code verification.

Contributions and Results

- Code auditing approach for efficient verification of SQLI and XSS defense features implemented in program source code.
- Algorithms for extracting the SQLI and XSS defense features.
- Prototype tool for automatic extraction of the defense features.
- Evaluation of the proposed approach based on seven Java-based web applications. In the experiments, using our proposed approach, the student auditors had to inspect only 7% of the total source code to effectively determine inadequate implementation of defense features and identify the SQLI and XSS vulnerabilities.

This chapter is organized as follows. Section 5.1 presents the concepts for extracting the defense features implemented in code. Section 5.2 presents the proposed code auditing approach. Section 5.3 illustrates the proposed approach with an example. Section 5.4 evaluates our approach with the aid of an existing vulnerability detection approach. Section 5.5 concludes the chapter.

5.1 Code-based Extraction of SQLI and XSS Defense Artifacts

In this section, we present our concepts on extracting the program artifacts, which are for securing the program from SQLI and XSS attacks. These concepts are built on modeling the possible code patterns of defensive coding methods. Through the empirical studies on many web applications, we observed that the following methods are generally implemented to prevent SQLI and XSS: (1) validation; (2) escaping; (3) filtering; (4) character set specifying. Escaping and filtering methods are also addressed as sanitization methods in the literature [36]. Before proceeding to discussing our concepts, we shall first introduce some terminologies that will be used throughout this chapter.

Since our objective is to assist auditors in their security auditing process, given a program, this work aims to identify essential set of nodes in the program and extract security information from them. Therefore, in addition to the terms and definitions of control flow and data flow relations given by Sinha et al. [140] (discussed in Section 2.2), we introduce the following definitions to be used in our auditing approach:

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

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

Definition 3. A variable v is *influenced* by a node u if and only if v is defined at u or v is defined at a node that is (transitively) data dependent on u .

In a web application, the input data submitted by an external user shall be addressed as *tainted data* because such input could be crafted to cause SQLI and XSS.

Definition 4. A node u at which *tainted data* submitted by user is referenced is called an *input node*.

We model the nodes at which inputs submitted through any external sources, such as HTTP request parameters—GET, POST, Header, and Cookie, as input nodes because such inputs are controlled by users. We also model the nodes, at which the data from database and session objects is accessed, as input nodes in order to prevent second order SQLI attacks [6] and stored XSS attacks [36] (as discussed in Session 2.1, second order attacks are basically conducted through the

input data stored into persistent data stores). A variable v influenced by an input node u is called a *tainted variable* as it could propagate tainted data.

A path through a directed graph is called a *1-path* if it follows any loop at most once (that is, if it does not repeat any loop). Let u be an input node in a CFG. An *input path* of u is a path from u to the exit node that does not pass through u again. An input path of u is called a *prime input path* of u if it iterates any loop at most once.

Definition 5. A node is called a *potentially vulnerable output node (pv-out node)* if it satisfies the following two conditions:

- it references at least one tainted variable, and
- it generates HTML output or accesses database

From the empirical studies on a handful of web applications, we have identified that nodes performing the following checks—data presence (null) check, string length check, string equality check, character containment check, data type check (e.g., integer, String, etc.); and nodes performing the following operations—string replacement or removal, data type transformation (e.g., from String to integer), character set specifying for client-server-database data transferring, and escaping (i.e., method call to an escaping library) on a tainted variable referenced in a pv-out node could sanitize the value of that variable. Therefore, we define the above checks and operations as *security checks* and *security operations* respectively; and the nodes performing them as *security nodes*. Based on the above observations, we classify the security nodes into four types—validation nodes, escaping nodes, filtering nodes, and char-set specifying nodes.

Before we provide the definitions of security nodes, we shall first introduce a variant of CFG that serves as a model for auditing the adequacy of SQLI and XSS defense artifacts.

Definition 6. Let N' be the subset of nodes in a CFG G that includes the following types of nodes:

- 1) Pv-out node, k .
- 2) All security nodes for k .
- 3) Input nodes which influence the variables referenced at k .

Let E' be the set of edges that connects the nodes in N' . For each unordered pair of nodes (n, m) in N' , if there is a path in G from n to m without passing through another node in N' , an edge (n, m) is included in E' ; if n is a predicate node or an exception node, the branch label ('True' or 'False') at n corresponding to that path is also included in the edge (n, m) . Thereby, the directed

graph $T = (N', E')$ is called the tainted-information flow graph (TIFG) for k , where N' and E' are its set of nodes and edges respectively.

According to *Definition 6*, for each pv-out node k , the TIFG T is constructed from a set of nodes selected from the CFG G of a program. The construction of T requires the application of *Definitions 1-5* discussed above. Basically it is constructed by removing the nodes in G that have no influence in tainting the data referenced in pv-out node. Thus, it models the information flows between the input nodes and a pv-out node. And this model provides essential information on the defense features implemented in a program.

In the following sub-sections, we shall provide methods for extraction of security nodes that make up the TIFG of a given pv-out node.

5.1.1 Characterizing Defense through Validation

Input validation is a traditional approach for handling external data in web applications. This method could reject invalid input immediately. It is typically used to ensure input data correctness. In today's web applications, data accuracy could also ensure data security; therefore, nodes in a CFG which implement this defense method should be extracted and checked for adequacy in defending against SQLI and XSS.

Let G be a CFG of a given web program. Let k be a pv-out node in G and v_k be a tainted variable referenced at k . Throughout the rest of the chapter, we shall address G , k , and v_k as CFG, pv-out node, and tainted variable referenced at k respectively. There may be more than one tainted variable referenced at k . It is common that the program uses predicate nodes or exception nodes to allow v_k to be operated at k if and only if the value of v_k satisfies the user interface specifications or some required conditions. Next, we provide a definition that characterizes such node pattern.

Definition 7. Let d be a predicate node or an exception node in a CFG and v_d be a variable referenced at d . The node d is called a *validation node* for k if the following properties hold:

- 1) Both v_k and v_d are influenced by the same input node (or nodes) u . That is, either both v_k and v_d represent the same variable or they are defined at nodes that are transitively data dependent on the same input node (or nodes).
- 2) A prime input path p of u that follows one branch of d passes through k but no prime input path p' of u that follows the other branch of d passes through k .

According to *Definition 7*, the presence of validation node for a pv-out node implies the existence of *security check* on an input before the input is referenced in the pv-out node because

validation node's properties ensure that only inputs satisfying certain condition checks would be referenced in subsequent pv-out nodes. Note that if d is an exception node, a branch from d leads to an immediate exit node or an exception handling node.

5.1.2 Characterizing Defense through Escaping

Although input validation may be used as a primary defense against all kinds of code injection attacks via inputs, validation methods may not be sufficient against all SQLI and XSS attacks. Therefore, for absolute prevention of code injection attacks, escaping (also called encoding) is often used to complement input validation. Escaping is a technique that ensures any special characters significant to a certain interpreter are just treated as data not as code. To prevent SQLI and XSS, proper escaping methods, such as SQL escaping, HTML entity escaping, URL escaping, and JavaScript escaping, need to be used according to the context in which the tainted data is referenced (i.e., according to the type of script parser interpreting the tainted data) [28, 107, 109]. Hence, this method will not work if the escaping scheme is inappropriate. For example, a developer may use SQL escaping scheme (i.e., special characters significant to SQL parser are escaped) on the tainted data before the data is passed into SQL statements and stored in a database. However, it is possible that the same data stored is later used in HTML outputs. But, since the SQL escaping scheme does not escape the special characters significant to the HTML interpreters, this causes XSS vulnerability. Therefore, nodes implementing defense through escaping need to be extracted and examined for adequacy.

Definition 8. A node in G is called an *escaping node* for k if it satisfies the following properties:

- 1) Node which is (transitively) data dependent on an input node and on which k is (transitively) data dependent.
- 2) Node that contains a method call to an escaping library.

In implementation-wise, escaping is a string replacement operation that replaces special characters with HTML equivalent characters (e.g., `<` with `<`) or database-specific escape characters (e.g., `'` with `\'`). However, instead of codifying this tedious task, web applications often deploy a standard escaping library and call its APIs to escape the tainted variables. Therefore, in *Definition 8*, escaping node is defined as the node which invokes an API call to an escaping library (e.g., OWASP's escaping libraries [110]) rather than the node (usually a set of nodes) which actually performs string replacement operations for escaping. As such, if the application performs escaping using custom code without API calls, this approach will not recognize the nodes

representing such program statements as escaping nodes. However, they shall still be extracted as filtering nodes according to the characterization of filtering nodes which is explained in the next sub-section.

5.1.3 Characterizing Defense through Filtering

Escaping could completely prevent SQLI and XSS. But it is required that correct escaping method is applied according to the context in which the tainted data is referenced. As the deployment and configuration of a standard escaping library is also typically required, some developers may not prefer this method. Instead, they would develop custom defense code using input filtering to prevent SQLI and XSS. Filtering is a technique that either removes or replaces malicious characters with non-malicious ones. Therefore, unlike input validation, filtering can be used to directly address SQLI and XSS. Among the discussed three defense methods, filtering method is the most common in web applications due to the above reasons. As such, inspection of filtering nodes is essential.

Such filtering if implemented in a web application is generally carried out by nodes in G that influence the tainted variables of pv-out node.

Definition 9. A node in G is called a *filtering node* for k if it satisfies the following properties:

- 1) Node which is (transitively) data dependent on an input node and on which k is (transitively) data dependent.
- 2) Node which is not an escaping node.

Using *Definition 9*, the sequence of nodes from input nodes to k representing the flow of tainted information possibly through a set of security operations can be extracted. Note that operations performed in the extracted filtering nodes may also include ordinary operations such as string concatenation (i.e., operations that are unintended for security purposes). However, we argue that all such nodes could help in understanding the structure of the filtering mechanism applied because these operations may also contribute to the definitions of the tainted variables of the pv-out node.

5.1.4 Characterizing Defense through Character Set Specifying

Web browsers and databases accept various character encoding schemes (e.g., UTF-8, UTF-16, and ISO-8859-1) when parsing a HTML document or a SQL query. It is extremely important that client-server and server-database communications always specify the character set to be used when

parsing a HTML document or a SQL query. If not specified, the browser may use the client's preferred (default) character set that may be different from the one used by the server program. The database may parse a SQL query with a character set different from the one used by the server program.

In such cases, it is possible to inject scripts that circumvent the above input validation, escaping, and filtering methods by encoding SQLI and XSS attack vectors with a different character set used by the server program. For example, a widely-known and detectable SQLI attack vector `1' OR '1'='1` could be encoded as `1' OR '1'='1` using hexadecimal encoding scheme to avoid detection. Especially, as web browsers are designed to offer best effort service, they would attempt to identify the encoding scheme used and interpret the encoded data, thereby often allowing encoded XSS scripts to run in the client's browsers. There are many instances where even the perfect validation, escaping, and filtering methods implemented in server programs have failed to prevent encoded SQLI and XSS attacks simply because server program does not recognize encoded malicious characters. And this can be easily avoided by ensuring that the same character set is used for all sides among server, client, and database.

Therefore, for auditing XSS defense methods, it is important to identify the presence of the node which implements this task in G . The following definition formalizes such node:

Definition 10. A node in G which sets the *charset* parameter defining a character encoding scheme to be used when parsing the data transmitted between server side and client/database sides is called a *charset specifying* node.

5.1.5 Security Checks and Operations

Web applications in general implement the combinations of defense methods described in the above sub-sections in order to avoid SQLI and XSS vulnerabilities. However, it is often difficult for auditors to identify these implementations via inspection of the whole chunk of code. Therefore, in this section, we provide the concepts for extracting the defense information through the use of TIFG.

From G , the TIFG T can be constructed according to *Definition 6*. Security nodes to be included in T could be extracted from G according to *Definitions 7-10*. The graph T provides an abstraction of G focused on the defense features implemented for k . It also shows all possible paths from input nodes to k . Consequently, it facilitates the extraction of security checks and

operations performed at different program paths leading to k . Auditors could then verify if every path from an input node to k is safe via inspecting T .

If a node in T is a security node except validation node, the operation performed on the tainted variable shall be extracted as a security operation. If it is a validation node, the condition check performed on the tainted variable shall be extracted as a security check. This information tells how the pv-out node is defended against SQLI and XSS attacks via malicious inputs. Hence, through examining this information, an auditor may determine the adequacy of SQLI and XSS defense implemented in code. The extraction method is described as a property in *Property 1*. This property can be proved directly from the definitions of security nodes (*Definitions 7-10*).

Property 1 – Security Checks and Operations of PV-Out Node. Let Ω be the set of 1-paths through T . The set of security checks and operations for the pv-out node k is $C_k = \{(X, C) \mid X \text{ is a path in } \Omega \text{ and it passes through a set of input nodes } I_x \text{ and security nodes } S_x \text{ for } k; \text{ and } C = \text{the set of tainted variables defined from } I_x \text{ and the set of security checks and/or operations performed at the nodes in } S_x\}$.

5.2 Auditing the Defense against XSS and SQLI Threats

Based on the concepts discussed in the previous section, we now present the proposed approach for auditing the SQLI and XSS defense features implemented in web applications. Our approach comprises of two phases—(1) Extraction of defense features (2) Auditing the defense. In the first phase, we deal with automatic extraction of all the possible defense features that could prevent SQLI and XSS attacks. In the second phase, the information recovered in the first phase is used to verify the adequacy of defense methods in preventing potential attacks.

5.2.1 Extraction of Defense Features

The pseudocode of the algorithms that presents the first phase of our approach is shown in Figure 5-1. In the following, the algorithms are explained in detail.

Given a web application program P , the algorithm in Figure 5-1a initially computes its CFG G . It then identifies the input nodes and the pv-out nodes as follows. According to *Definition 4*, the input nodes I are extracted by identifying the nodes in G which reference data from HTTP request parameters (GET, POST, Header, and Cookie) and database. According to *Definition 5*, the pv-out nodes O are extracted through data dependency analysis between input nodes and HTML output nodes.

Algorithm extractDefense(program P)
Output: The set Π of security operations and checks for each potentially vulnerable output statement in P .

```

begin
1  initialize both  $I$ ,  $O$ ,  $V$ ,  $E$ ,  $F$ , and  $\Pi$  to empty sets
2  compute  $G :=$  the CFG of  $P$ 
3  compute  $I :=$  the set of input nodes in  $G$ 
4  compute  $O :=$  the set of pv-out nodes in  $G$ 
5  If ( $O$  is not empty) then
6      compute  $c :=$  charset specifying node in  $G$ 
7      include  $c$  in  $\Pi$ 
8      For (each pv-out node  $k$  in  $O$ ) do
9          compute  $V :=$  the set of validation nodes for  $k$  in  $G$ 
10         compute  $E :=$  the set of escaping nodes for  $k$  in  $G$ 
11         compute  $F :=$  the set of filtering nodes for  $k$  in  $G$ 
12          $T :=$  computeTIFG( $G$ ,  $I$ ,  $V$ ,  $E$ ,  $F$ ,  $k$ )
13          $\Pi_k :=$  computeDefense( $T$ )
14         include  $\Pi_k$  in  $\Pi$ 
15     endFor
16 endIf
17 return  $\Pi$ 
end

```

(a)

Algorithm computeTIFG(CFG G , input nodes I , validation nodes V , escaping nodes E , filtering nodes F , pv-out node k)
Output: Tainted-information flow graph of k

```

begin
1  initialize  $N'$  and  $E'$  to empty sets
2  include  $k$ ,  $I$ ,  $V$ ,  $E$ ,  $F$  in  $N'$ 
3  For (each unordered pair of nodes  $(n, m)$  in  $N'$ ) do
4      If (there is a path  $p$  in  $G$  from  $n$  to  $m$  &
           $p$  does not pass through another node in  $N'$ ) then
5          create an edge  $e := (n, m)$ 
6          If ( $n$  is a predicate node) then
7              include the branch label of  $p$  at  $n$  in  $e$ 
8          endIf
9          include  $e$  in  $E'$ 
10     endIf
11 endFor
12  $T := (N', E')$ 
13 return  $T$ 
end

```

(b)

Algorithm computeDefense(TIFG T for k)
Output: $\{(X, C) \mid C \text{ is the set of security checks and/or operations performed for } k \text{ along a program path } X\}$

```

begin
1  compute  $\Omega :=$  the set of 1-paths through  $T$ 
2  For (each path  $X$  in  $\Omega$ ) do
3      compute  $C :=$  the set of security checks and/or operations performed at the security nodes
          through which  $X$  passes through
4      include  $(X, C)$  in  $C_k$ 
5  endFor
6  return  $C_k$ 
end

```

(c)

Figure 5-1. (a) Algorithm for extracting SQLI and XSS defense features from code. (b) Algorithm for constructing TIFG. (c) Algorithm for computing defense information.

Next, the algorithm identifies charset specifying node c in G according to *Definition 10*. This task requires pattern matching. That is, the node patterns (language-specific implementation patterns) of SQL statement and HTML output statement that specify the character set to be used respectively (for example, see statement 1 and 2 in Figure 5-2) are matched against the nodes in G . Then, for each pv-out node k in G , the other three types of security nodes—validation nodes, escaping nodes, filtering nodes, are identified according to *Definitions 7-9*. Validation and filtering nodes are extracted using the following techniques: (1) control and data dependency analysis of the nodes between k and input nodes; (2) identification of the security checks and operations based on their language-specific implementation methods (e.g., character replacement operation at a filtering statement is identified when the statement invokes a function `'taint_var.replace()'` where `taint_var` is an tainted variable). Escaping nodes contain method calls to an escaping library according to its definition; therefore, this type of nodes is extracted using the following techniques: (1) control and data dependency analysis of the nodes between k and input nodes; (2) pattern matching to determine if a node contains a method call to an escaping library. For this pattern matching, our algorithm requires that user specifies the names of the escaping methods used in the application under test.

Based on G and the security nodes extracted above, the algorithm in Figure 5-1b constructs the TIFG for each pv-out node according to *Definition 6*. Based on T , the algorithm in Figure 5-1c computes the set C_k of security checks and operations for k according to *Property 1*. The computations are as follows:

- The set Ω of 1-paths through T is computed using depth first search algorithm. During the path traversal, any loop construct is traversed only once.
- For each path X in Ω , as X passes through a set of input nodes and security nodes for k and also k , based on *Property 1*, we include the variables defined in input nodes as a set of tainted variables, and the condition checks (including the branch label `'true'/'false'`) at validation nodes for k and the operations performed at filtering nodes and escaping nodes for k as a set of security checks and operations for k in C_k .

Note that in TIFG, statements that influence the data referenced in k are included in general. The process of extracting such statements is similar to program slicing. However, in opposing to program slicing, it does not extract all the statements that affect the control flow of k . Instead, it only extracts the predicates that reference the same input variables as k . As such, TIFG is designed to include all statements relevant to security. Yet, these statements shall only be a subset of

statements produced by slicing. More importantly, TIFG focuses on providing the flow of information through various paths from input nodes to k .

5.2.2 Auditing the Defense

The second phase of the proposed method is to examine whether the defense features extracted (in the first phase) for each pv-out node is sufficient for preventing SQLI and XSS attacks. From the fact that SQLI and XSS vulnerabilities are commonly found in many web applications, it is clear that this task is an intricate one for auditors. The two main difficulties that auditors may face is (1) among the defensive coding methods— character set specifying, validation, filtering, and escaping, web developers may employ each individual method or a combination of all these methods for the defense against SQLI and XSS; (2) the effectiveness of these defensive coding methods may not be consistent across web browsers due to anomalous web browser parsing quirks. Hence, the auditor must take all these alternatives into considerations while verifying the adequacy of those methods. The use of TIFG and the defense information extracted from TIFG mitigate these problems because auditors could then focus on a smaller and directed set of information. To further aid this process, next, we shall propose some guidelines for manual audition of the features extracted from TIFG.

Let T be a TIFG for k . Let Ω be the set of 1-paths through T .

Rule 1: If T does not contain a charset specifying node, then k is subjected to encoded attacks even in the presence of other security nodes. Thus, k is definitely vulnerable and the implementation of such node is recommended (it is possible to define character set encoding in configuration files, usually XML files, in application server and database server rather than in programs. Therefore, in the absence of charset specifying node in a program, auditor may have to further inspect the configuration files, but this topic is out of our scope). On the other hand, if such node is implemented, it can be concluded that malicious characters such as $<$ could not be encoded with character sets different from the one used by server program's sanitization functions.

Rule 1 holds the highest priority among the proposed rules. The auditor should further apply the following *Rules 2-6* only after applying *Rule 1*. That is, if T does not contain a charset specifying node, k is definitely vulnerable and the following rules do not hold anymore. Therefore, the following rules assume that k is not concluded as vulnerable according to *Rule 1*.

Rule 2: Let the variable v_k of k be influenced by an input node u . If there is no security node d in a path of Ω that references a variable also influenced by the same input node u , then k is

definitely vulnerable to SQLI/XSS attacks (the type of attack is dependent on the type of pv-out node) through manipulating the value of v_k .

Rule 3: Let the variable v_k of k be influenced by an input node u . If there are input validation nodes in T and the following conditions are met, then k is definitely not vulnerable to SQLI/XSS attacks:

- 1) An input validation node in a path of Ω performs numeric data type check on the variable also influenced by the same input node u .
- 2) Condition 1 is true for all the paths in Ω .
- 3) Condition 1 and 2 are true for all the tainted variables referenced in k .

Rule 4: Let the variable v_k of k be influenced by an input node u . If there are filtering nodes in T and the following conditions are met, then k is definitely not vulnerable to SQLI/XSS attacks:

- 1) A filtering node in a path of Ω performs transformation to numeric data type on the variable also influenced by the same input node u .
- 2) Condition 1 is true for all the paths in Ω .
- 3) Condition 1 and 2 are true for all the tainted variables referenced in k .

Rule 5: Let the variable v_k of k be influenced by an input node u . If there are escaping nodes in T and the following conditions are met, then k is definitely not vulnerable to SQLI/XSS attacks:

- 1) An escaping node in a path X of Ω adopts the escaping scheme that is appropriate to the context in which the tainted data is referenced. And there is no other escaping node in X (to avoid double escaping).
- 2) The escaping scheme applied is consistent across all the paths in Ω (i.e., condition 1 is true for all other paths of Ω).
- 3) Condition 1 and 2 are true for all the tainted variables referenced in k .

On the other hand, if any of the above conditions is not met, the escaping method implemented is probably not adequate and thus, k is probably vulnerable to SQLI/XSS attacks. In this case, the auditor is to further check the adequacy of other security nodes to conclude the vulnerability.

Rule 6: Let the variable v_k of k be influenced by an input node u . If the following conditions are met, then k is probably not vulnerable to SQLI/XSS attacks:

- 1) A security node in a path of Ω invokes a function that either sanitizes the variable also influenced by the same input node u or checks if the variable also influenced by the same

input node u satisfies certain requirements; and the function is commonly accepted as ‘adequate’ by the community.

- 2) Condition 1 is true for all the paths in Ω .
- 3) Condition 1 and 2 are true for all the tainted variables referenced in k .

The reasoning for *Rules* 3 and 4 is that SQLI and XSS can only be performed through injection of string-type data as long as the program specifies a character set (*Rule* 1). Therefore, ensuring that tainted variables only contain numeric values is definitely adequate for preventing SQLI and XSS attacks. In applying *Rule* 5, it is important to ensure that there is no double escaping performed on the same data even if the escaping scheme is appropriate because it would affect the intended output. The reasoning for *Rule* 6 is that if the application adopts a sanitization function widely accepted as ‘adequate’, it is very likely that the function is correctly implemented. Therefore, debugging of such function (i.e., tracing and checking of the function) may not be required. But the auditor may still need to check if it is properly adopted by the developer.

The above rules serve as a guideline for auditors in checking the adequacy of defense artifacts extracted from T . However, these rules may not be sufficient to aid the whole auditing process. Therefore, if the security nodes in T do not conform to the conditions in *Rules* 1-6, then the auditor is to carry out the thorough examination of security checks and operations performed on the tainted variables for each path in Ω of T . He is to check if the validation checks performed at validation nodes conform to user interface specifications and determine their adequacy in preventing SQLI and XSS attacks. Similarly, from the information of security operations performed at filtering and escaping nodes, he is to examine how these operations contribute to the defense against SQLI and XSS. For broader and more comprehensive guidelines to prevent SQLI and XSS, auditors may refer to the audition guidelines provided by security experts such as OWASP [107, 109] and RSsnake [118, 119].

5.3 An Example Illustrating the Proposed Approach

In this section, we illustrate the proposed code auditing approach with an example program shown in Figure 5-2.

Figure 5-2a shows an example of a Log-In authentication program and Figure 5-2b shows the CFG representing the program. In Figure 5-2b, the nodes 4, 6, 7, and 14 are input nodes. Variables `uname`, `pwd`, `rs`, `name`, and `htmlOut` are tainted variables as their values are influenced by the input nodes. The nodes 14, 17, and 20 are pv-out nodes as these nodes reference tainted variables, and generate HTML outputs or access database. Nodes 1, 13, and 21 are not pv-out nodes as they do

In Figure 5-2b, the pv-out nodes 14, 17 and the predicate node 12 commonly reference the tainted variable `uname` influenced by the input nodes 4 and 6. And the security checks—*null check* and *string length check* are performed at node 12. A prime input path (4, 5, 6, 7, 8, 9, 12, 14, 15, 16, 17, . . . , end) that follows the false branch (12, 14) of node 12 passes through node 14 and node 17. No prime input path that follows the true branch (12, 13) of node 12 passes through node 14 and node 17. Therefore, node 12 is the validation node for the pv-out nodes 14 and 17. Similarly, predicate node 15 is the validation node for pv-out node 17 (because the variable `rs` is influenced by the input nodes 4, 6, and 7). Predicate node 15 and pv-out node 20 reference the tainted variables `rs` and `htmlOut`, which are influenced by a common input node 14. Thus, node 15 is the validation node for node 20. Similarly, node 12 is the validation node for node 20.

Node 18 is an escaping node for the pv-out node 20 because (1) it is data dependent on the input node 14, and node 20 is data dependent on it; (2) it calls an encoding library. Whereas, nodes 9 and 16 are filtering nodes for node 17 because (1) node 9 is data dependent on the input node 7, and node 16 is data dependent on the input nodes 4 and 6, and (2) the pv-out node 17 is data dependent on both nodes 9 and 16. Similarly, it can be deduced that nodes 9, 18, and 19 are filtering nodes for the pv-out node 20.

Nodes 1 and 2 are charset specifying nodes as the nodes define the character set to be used in client side and database respectively.

As shown in Figure 5-2c, we constructed the TIFGs for the pv-out nodes 14, 17, and 20. According to *Property 1*, $C_{14} = \{(\text{tainted variables } \text{uname} \text{ and } \text{pwd} \text{ defined from node 4 and node 7 respectively, } \text{charset=UTF-8} \text{ at nodes 1 and 2, } \text{Integer.parseInt(pwd)} \text{ at node 9, } !(\text{uname} == \text{null} \mid \mid \text{uname.length()} > 10) \text{ at node 12}), (\text{tainted variables } \text{uname} \text{ and } \text{pwd} \text{ defined from node 6 and node 7 respectively, } \text{charset=UTF-8} \text{ at nodes 1 and 2, } \text{Integer.parseInt(pwd)} \text{ at node 9, } !(\text{uname} == \text{null} \mid \mid \text{uname.length()} > 10) \text{ at node 12})\}$ is the set of tainted variables, and security checks and operations performed for the pv-out node 14.

Similarly, $C_{17} = \{(\text{tainted variables } \text{uname} \text{ and } \text{pwd} \text{ defined from node 4 and node 7 respectively, } \text{charset=UTF-8} \text{ at nodes 1 and 2, } \text{Integer.parseInt(pwd)} \text{ at node 9, } !(\text{uname} == \text{null} \mid \mid \text{uname.length()} > 10) \text{ at node 12, } (\text{rs} == \text{null}) \text{ at node 15, } \text{uname.replace("<", "")} \text{ at node 16}), (\text{tainted variables } \text{uname} \text{ and } \text{pwd} \text{ defined from node 6 and node 7 respectively, } \text{charset=UTF-8} \text{ at nodes 1 and 2, } \text{Integer.parseInt(pwd)} \text{ at node 9, } !(\text{uname} == \text{null} \mid \mid \text{uname.length()} > 10) \text{ at node 12, } (\text{rs} == \text{null}) \text{ at node 15, } \text{uname.replace("<", "")} \text{ at node 16})\}$ is the set of tainted variables, and security checks and operations performed for the pv-out node 17.

Finally, $C_{20} = \{(\text{tainted variables } \text{uname}, \text{ pwd}, \text{ and } \text{rs} \text{ defined from nodes 4, 7, and 14 respectively, charset=UTF-8 at nodes 1 and 2, Integer.parseInt(pwd) at node 9, }!(\text{uname} == \text{null} \text{ || } \text{uname.length()} > 10) \text{ at node 12, (rs == null) at node 15, } \text{htmlEncode(rs.getString("name")) at node 18}), (\text{tainted variables } \text{uname}, \text{ pwd}, \text{ and } \text{rs} \text{ defined from nodes 6, 7, and 14 respectively, charset=UTF-8 at nodes 1 and 2, Integer.parseInt(pwd) at node 9, }!(\text{uname} == \text{null} \text{ || } \text{uname.length()} > 10) \text{ at node 12, (rs == null) at node 15, } \text{htmlEncode(rs.getString("name")) at node 18})\}$ is the set of tainted variables, and security operations performed for node 20.

Note that for each of these pv-out nodes, the set of tainted variables, and security checks and operations consists of exactly two elements since their corresponding TIFGs have two 1-paths.

TIFG may also include additional information not related to security defense. For example, node 19 in the TIFG of the pv-out node 20 performs a string concatenation operation that is not intended for security. However, such information could still be useful for auditing process (e.g., information extracted from node 19 can be used to infer the type of escaping mechanism required to prevent XSS).

The following discusses how an auditor could check for adequacy of the defense features through the use of information extracted from TIFG.

For the pv-out node 14, from the information extracted from the TIFG, an auditor could infer according to the audition guidelines provided in Section 5.2 that (1) the character set specification at node 2 prevents encoded SQLI attacks (*Rule 1*); (2) the data type transformation at filtering node 9 un-taints the variable `password` and thus no script injection via that variable is possible (*Rule 4*); (3) the data presence check and string length check on the variable `uname` at validation node 12 ensures that no SQLI attack pattern containing more than 10 characters could reach node 14. By summarizing this information, an auditor could conclude that node 14 is still vulnerable because an SQLI attack via `uname` crafted with a maximum of 10 characters (e.g., `uname= 1' OR 1=1`) would exploit it. Interested readers may refer to [6, 118] for detailed exploitation mechanisms.

Similarly, for the pv-out node 17, an auditor could infer that (1) the character set specification at node 1 prevents encoded XSS attacks (*Rule 1*); (2) the data type transformation at filtering node 9 un-taints the variable `password` and thus no script injection via that variable is possible (*Rule 4*); (3) the data presence check and string length check on the variable `uname` at validation node 12 ensures that no XSS attack pattern containing more than 10 characters could reach node 17; (4) however, the data presence check on the variable `rs` at node 15 may not serve any security

purpose; (5) the character replacement of the variable `uname` at node 16 ensures that XSS attack patterns formed with character ‘<’ are not allowed. Thus, summarizing this information, the auditor could conclude that node 17 is not vulnerable because it is required to use ‘<’ character to create a successful XSS attack via the parameter referenced in an HTML body [109, 119] and the use of character encoding to conceal ‘<’ is also not possible due to (1).

For the pv-out node 20, the auditor has to identify the HTML context in which the tainted variable `name` is referenced and determine if an escaping method ‘`htmlEncode`’ applied is appropriate (*Rule 5*). From node 19, the auditor may infer that the tainted data is referenced as a value of an HTML attribute. If ‘`htmlEncode`’ is appropriate for that context, the auditor may conclude that node 20 is safe from XSS. Note that many web applications tend to commonly apply HTML entity encoding method for all the tainted data referenced in different HTML contexts. If the above method ‘`htmlEncode`’ is an HTML entity encoding method, it has to be concluded that node 20 is vulnerable as entity encoding is not a perfect solution for this context (see appropriate escaping methods for different HTML contexts in [109]).

5.4 Evaluation

We evaluated our approach on seven open source Java-based web applications: *Employee Directory*, *Bookstore*, *Events*, *Classifieds*, *Portal*, *PersonalBlog*, and *JOrganizer*. The first five test subjects were obtained from GotoCode [44] and the other two test subjects were obtained from SourceForge [143]. Table 5-1 shows the statistics of these test subjects. The lines of code (LOC) counts shown in Table 5-1 do not include library classes. The last two columns show the number of program statements that are vulnerable to SQLI and XSS attacks respectively. We did not conduct SQLI vulnerability auditing tests for the two test subjects—*PersonalBlog* and *JOrganizer* as we do not have the vulnerability information regarding to SQLI on these two applications.

As we claimed that the proposed approach would be useful in aiding the existing vulnerability detection approach, in the evaluation, we also used an existing vulnerability detection approach—Livshits and Lam’s approach [82]. We compared our work with Livshits and Lam’s work based on the same seven test subjects above. Our evaluation setup is justified because (1) our test subjects have also been benchmarked by related major approaches [48, 49, 85, 144] and (2) as shown in Table 5-1, the test subjects’ sizes range from small to large scales and they differ in functionality.

In literature, state-of-the-art static analysis-based approaches for detecting SQLI and XSS vulnerabilities include approaches proposed in [61, 82, 165]. We chose Livshits and Lam’s approach [82] for comparison because their tool also targets Java-based applications like ours. The

other two state-of-the-art static-based approaches [61, 165] target PHP-based applications. Although there are more sophisticated and accurate vulnerability detection approaches like Kiežun et al.’s concolic execution technique [68], those approaches require dynamic analysis, which has scalability issues (see Chapter 7.4). Both our work and Livshits and Lam’s approach only apply static analysis.

SQLI and XSS vulnerability auditing experiments were conducted in three steps—(1) we applied Livshits and Lam’s approach to simulate the automatic process of locating the potential SQLI and XSS vulnerabilities in the test subjects; (2) on the same test subjects, our prototype tool *WAVDE* (see Section 5.4.1) was applied to extract the defense features implemented for every HTML output and SQL statement referencing the tainted data. The extracted features were then audited to also identify the vulnerabilities as in the first step but using the proposed auditing technique; (3) we verified the results of both approaches and evaluated our proposed technique.

In this evaluation, we aim to answer the following two research questions:

RQ1. Is the code auditing approach really necessary despite the availability of current static analysis-based vulnerability detection approaches?

RQ2. Is our approach effective in extracting SQLI and XSS defense features and feasible in examining them to determine their adequacy?

Table 5-1. Statistics of the test subjects

Test Subject	Description	LOC	#Servlets	#Stmt vuln. to SQLI	#Stmt vuln. to XSS
Employee Directory	Directory system for tracking employee	3,035	10	8	44
Bookstore	Online bookstore	9,551	28	17	143
Events	Event management system	3,818	13	10	20
Classifieds	Advertisement and shopping system	5,745	19	17	19
Portal	Portal for a club	8,803	28	24	143
PersonalBlog	Blogging system	17,149	31	-	1
JOrganizer	Contact and appointment management system	31,897	153	-	21
Total		79,998	282	76	391

5.4.1 Prototype Tool

We implemented our proposed approach in a prototype tool called Web Application Vulnerability Defense Extractor (*WAVDE*) through the use of *Soot* [142]. *Soot* provides Java bytecode analysis framework upon which control and data dependency analysis is implemented in

WAVDE. Our tool fully automates the first phase of our approach, which is extracting defense features (Section 5.2.1).

The architecture of *WAVDE* is shown in Figure 5-3. It consists of three modules: program analyzer, defense feature miner (DF miner), and TIFG. *Program analyzer* uses *Soot*'s APIs to analyze Java Servlet programs. It takes the class files of a program as input and builds a CFG of the program. *DF miner* includes three components: validation extractor, filter extractor, and escape extractor. By performing control and data flow analysis of a given CFG, *DF miner* first computes the set of input nodes and pv-out nodes. Security nodes are then extracted by the three extractors according to the definitions of security nodes. If the application under test adopts a standard escaping library, the user has to specify the names of the escaping APIs used via a user interface so that the escape extractor recognizes them as escaping nodes. For each pv-out node, *TIFG module* first constructs its respective TIFG using the security nodes extracted by *DF miner*. The graph is constructed according to *Definition 6*. Next, the module recovers the set of security checks and operations performed along 1-paths of the computed TIFG. All the information recovered is printed in a report and viewable via a user interface. For more information about the tool, please refer to our website [116].

Figure 5-3. Architecture of the prototype tool, *WAVDE*

5.4.2 Experiment

We requested four post-graduate students, who have taken Software Engineering course and scored ‘A’ Grade, to assist in this experiment. Two weeks training on SQLI and XSS injection techniques and static program analysis techniques was provided. Two students learned how to apply Livshits and Lam’s approach and another two students learned how to use WAVDE and apply the proposed approach. Before the experiments begin, we manually inspected the source code of each test subject and identified the actual SQLI and XSS vulnerabilities in the test subjects. From our manual inspections, we observed that these vulnerabilities in the test subjects are mainly caused by improper input sanitization (*Employee Directory*, *Bookstore*, *Events*, *Classifieds*, *Portal*) or failure to identify all input sources (*PersonalBlog*, *JOrganizer*). The last two columns of Table 5-1 show the numbers of actual SQLI and XSS vulnerabilities we identified. Thereafter, the experiment was conducted in the following three steps.

The first step of the experiment is to locate the (potentially) vulnerable program statements in the test subjects using an existing approach that could automate this task in a real scenario. However, we conducted Livshits and Lam’s approach analytically since their prototype tool is not available to us. Basically, Livshits and Lam’s approach tracks the flow of data from input sources via derivation descriptors into sensitive sinks (SQL and HTML output statements). If the data from a specified input source passes through a specified derivation descriptor, it implies that the data is still tainted. Thus, the sink which references that data is vulnerable. On the other hand, if the data from a specified input source passes through a function or an operation not specified as derivation descriptor, the sink referencing that data is not vulnerable. Therefore, to avoid any false negative cases, Livshits and Lam’s technique requires adequate vulnerability specification as an input. For this experiment, we assumed that the auditor generates a perfect vulnerability specification that reflects all the functions (described in terms of sources, sinks, and derivation descriptors) we encountered when inspecting the test subjects. The students simulated Livshits and Lam’s analysis by performing the following two steps: (1) analyze source code and find code patterns that match the vulnerability specification given by us; (2) report the SQL statement or HTML output statement as vulnerable if any match is found. The two students applied the technique independently on the test subjects and their results were reconciled before taken as a final result.

The second step is to apply the proposed approach on the same test subjects. The other two students were given this task. We provided them with (1) access to WAVDE; (2) six rules proposed in Section 5.2.2 as a primary guideline for audit trial; and (3) SQLI and XSS prevention guidelines obtained from OWASP [107, 109] and RSNAKE [118, 119] as further references. The students first

ran the prototype tool *WAVDE* on the test subjects to automatically extract the defensive statements implemented for each potentially vulnerable statement (pv-out node in a CFG) and to obtain the systematic reports on the security checks and operations performed in those extracted statements. Following the guidelines provided, the students independently audited the extracted defense features implemented for each pv-out statement (identified by *WAVDE*). If the student determined that an appropriate defense against SQLI or XSS attacks is not in place, the corresponding pv-out statement was reported as ‘vulnerable’ or vice-versa. Results from the two students were reconciled before taken as a final result.

The final step is to evaluate the two final results from step 1 and 2. Using the information obtained from inspecting the test subjects ourselves, we verified the results reported by the students. Results on auditing the vulnerability of SQL statements are shown in Table 5-2. Results on auditing the vulnerability of HTML output statements are shown in Table 5-3.

5.4.3 Result

To answer the two research questions raised, we analyzed the statistics of the experiment results.

RQ1: As shown in Table 5-2 and Table 5-3, in total, Livshits and Lam’s approach reported 279 (58+221) false positives, which accounted for 37% ($279/(134+612)$) of the total reported vulnerable statements (Livshits and Lam’s approach also produced 29% (12/41) error rate in their own evaluation [82]). In the experiments using the proposed approach, from the output extracted automatically, both students identified all the vulnerability cases accurately without any false positive cases. We observed that false positive cases reported by Livshits and Lam’s approach mainly arise from two sources: (1) we specified some of the string operations (such as concatenation, replacement, and substring) as derivation descriptors because we assumed that those operations still propagate tainted data; (2) some inputs are properly handled through the use of complex control-flow constructs. In the first case, the assumption is valid and important as it avoids false negative cases. However, it causes some false positive cases for Livshits and Lam’s approach in this experiment because (a combination of) the string operations applied in some program statements result in properly sanitized inputs. By contrast, using the proposed manual auditing approach, the student auditors were able to correctly identify those properly sanitized inputs. The second case is because Livshits and Lam’s analysis does not consider control-flow constructs. By contrast, our approach extracts both data dependency and control dependency statements for verification and auditing.

Table 5-2. Results on SQLI vulnerability auditing

© 2012 IEEE

Test Subject	Livshits and Lam's Approach		Proposed Approach			
	#Vulnerable statements reported	#False positives	Defense features extracted by WAVDE (LOC)	Extraction time by WAVDE (s)	#Vulnerable statements identified from manual code audits	#False positives
Emp. Dir.	12	4	189	21	8	0
Bookstore	38	21	461	29	17	0
Events	16	6	174	12	10	0
Classifieds	34	17	308	24	17	0
Portal	34	10	466	25	24	0
Total	134	58	1,598	111	76	0

Table 5-3. Results on XSS vulnerability auditing

Test Subject	Livshits and Lam's Approach		Proposed Approach			
	#Vulnerable statements reported	#False positives	Defense features extracted by WAVDE (LOC)	Extraction time by WAVDE (s)	#Vulnerable statements identified from manual code audits	#False positives
Emp. Dir.	54	10	459	49	44	0
Bookstore	187	44	1,636	117	143	0
Events	65	45	510	47	20	0
Classifieds	89	70	751	78	19	0
Portal	194	51	1,752	155	143	0
PersonalBlog	1	0	379	224	1	0
JOrganizer	22	1	152	345	21	0
Total	612	221	5,639	1,015	391	0

A simple example reflecting the second case is shown in the following code taken from a test subject—*Events* (the code is simplified).

```
1 if(isNumber(request.getParameter("sorted"))) {  
2     String sSorted = request.getParamater("sorted");  
    . . .  
3     out.println("<table><tr> . . . "+ sSorted);
```

Intuitively, the method `isNumber` checks if the method argument contains only numeric data. Livshits and Lam’s approach does not recognize that the tainted variable `sSorted` is safe due to the validation check at line 1. This is because `sSorted` does not actually pass through the sanitization function `isNumber`. Thus the approach will report statement 3 as vulnerable, resulting in a false positive case. By contrast, the student auditors were able to conclude (from the validation check at line 1 extracted by our tool, *WAVDE*) that the variable `sSorted` at line 2 is safe. These cases highlight the strength of a code auditing technique. The false positive cases resulting from an existing static analysis approach can be rectified by our proposed technique.

Livshits and Lam’s approach did not produce false positives (except one) in auditing the test subjects *PersonalBlog* and *JOrganizer*. This is because the two subjects only use adequate sanitization procedures and numeric type casting functions in checking the inputs. Thus, there was no false positive due to validation checks. All the actual vulnerabilities of these two subjects only arise from the total absence of checks probably due to the misidentification of input sources. Hence, both sets of students who used Livshits and Lam’s approach and our proposed approach easily identified those vulnerabilities.

Livshits and Lam’s approach did not produce any false negative cases as we assumed that the auditor is able to generate an adequate vulnerability specification. This may not be the case in practice. The auditor may fail to specify a function or an operation (as derivation descriptor) that still propagates the tainted data. In such cases, the propose code auditing approach would be useful in identifying the vulnerable statements that might be missed by Livshits and Lam’s approach because the proposed method extracts the defense features implemented for every SQL and HTML output statement which references tainted data.

In summary, at least in this evaluation, the results show that our proposed auditing approach is really necessary for rectifying the inadequacy of defense features implemented for preventing SQLI and XSS attacks, and could be useful in aiding existing vulnerability detection approaches, thus answering our first research question.

RQ2: In total, 7,237 (1,598+5,639) LOC were extracted as SQLI and XSS defense features as shown in Table 5-2 and Table 5-3. Through the manual inspection on the test subjects, we have confirmed that the *WAVDE* tool completely extracted all the SQLI and XSS defense features implemented. SQLI defense features made up of 5% (1,598/30,952) and XSS defense features made up of 7% (5,639/79,998) of the total LOC. Since code auditing is really important and should not be avoided (as we explained above), it is clearly much more feasible for the auditors to audit this amount of code than the whole chunk of code.

Table 5-2 and Table 5-3 also show the time spent on extracting those defense features by *WAVDE* (as Livshits and Lam's tool is not available to us, we could not make a comparison). The results show that the extraction time is insignificant. The maximum time spent was on the test subject *JOrganizer* and *WAVDE* took only about 6 minutes. Also, in performing manual audits on the extracted defense features, the student auditors were able to successfully complete the process in a given amount of time, which is 2 hours for each test subject.

Hence, from these results, we can conclude that our proposed auditing technique is effective in extracting SQLI and XSS defense features and feasible in examining them to determine their adequacy, thus answering our second research question.

5.4.4 Limitation

Unlike automated vulnerability detection methods, our method is a comprehensive auditing approach to audit the defense against SQLI and XSS implemented in a system. Therefore, manual work is required and an auditor is required to understand and follow the guidelines (Section 5.2.2) for preventing SQLI and XSS. However, as we discussed earlier, though existing vulnerability detection methods are automated, their reports may contain both false-positives and false-negatives. Therefore, these methods cannot avoid code verification need and our method would complement them.

The extracted defense features may also include statements that do not serve any security purposes. In this case, the auditor has to identify the appropriate portion that serves as the defense against SQLI and XSS attacks.

The test subjects used in our experiment may not reflect commercial or industrial applications. The code auditing process may not also reflect a real auditing environment as the students are not expert auditors. However, our experiments show that even students who lack experiences are able to carry out our method with effective results. Therefore, we believe that our approach would still be practical under real-world settings.

5.5 Conclusion

In this chapter, we have presented a code auditing approach that systematically recovers SQLI and XSS defense artifacts implemented in web programs. We also provided guidelines to aid the auditing of recovered defense artifacts.

Although SQLI and XSS vulnerabilities could be automatically identified by existing vulnerability detection approaches, in practice, actual vulnerable cases still has to be confirmed through manual inspection. To fix the vulnerabilities, the auditors would also need to verify the shortcomings of the defenses implemented in programs. Therefore, current methods cannot avoid the code auditing need. But, audition on the whole source code would be labor-intensive while audition on the limited information provided by existing approaches would be inadequate. Our code auditing technique addresses this problem by providing comprehensive and precise information on the defense features implemented in systems.

Our evaluation has shown that the proposed approach is feasible for the sizes of the test subjects and useful in assisting to vulnerability auditing process due to its effectiveness in extracting all the statements relevant to SQLI and XSS defenses. We have also observed that existing vulnerability detection approaches could be combined with our work to automate the detection of potential vulnerabilities. This would enhance and speed up our manual code auditing process.

Chapter 6

XSS VULNERABILITY REMOVAL⁷

Vulnerability if existed in a program has to be removed. Vulnerability mitigating techniques such as runtime attack prevention might be effective at intercepting security attacks during runtime; but, these techniques do not analyze source code and focus on removing vulnerabilities that exist in program source code. As more and more sophisticated vulnerability exploitation techniques are being discovered, any un-removed vulnerabilities could become exploitable at times.

On the other hand, defensive coding practices could, in principal, ensure that a program is free from vulnerabilities. The known defensive coding approach for effective prevention of XSS vulnerabilities is to escape all the user inputs used in HTML documents according to the contexts in which these inputs are referenced (e.g., *JavaScript* context if the input is used in a JavaScript code; *HTML element* context if the input is used in an HTML element). Escaping in the context of XSS is the process of transforming the characters that have special meanings to a client-script interpreter into the representations such that the special meanings are removed [97]. However, this method if performed manually is prone to human errors and hard to be enforced into existing web applications. Therefore, automation of this task would be beneficial.

In this chapter, we present an automated approach that statically removes XSS vulnerabilities from program source code. This approach is different from existing approaches which mainly concern with either locating XSS vulnerabilities in program source code or preventing real time XSS attacks. The proposed method consists of two phases: (1) XSS vulnerability detection and (2) XSS vulnerability removal. XSS vulnerability detection phase identifies potential XSS vulnerabilities in the program source code using static analysis. XSS vulnerability removal phase first determines the context of each user input referenced in the identified potential vulnerabilities.

⁷ This chapter is subject to Elsevier B.V. Copyright 2012. The original source is in DOI: 10.1016/j.infsof.2011.12.006

It then secures the potential vulnerabilities by applying the appropriate escaping methods using an escaping library provided by OWASP [110].

Contributions and Results

- Automated approach for effective removal of XSS vulnerabilities from program source code.
- Prototype tool called *saferXSS* that implements the proposed approach and provides full automation.
- Evaluation of proposed XSS removal approach using *saferXSS*. Experiments have been conducted on five Java-based web applications. In the experiments, the approach was effective in securing all the XSS vulnerabilities found in the test subjects.

This chapter is organized as follows. Section 6.1 discusses adequate coding rules to follow for prevention of XSS. Section 6.2 presents our automated XSS removal technique built on those XSS prevention rules. Section 6.3 evaluates the proposed technique. Section 6.4 concludes this chapter.

6.1 XSS Prevention Rules

We discussed in Chapter 2.1.2 that XSS attacks are carried out by injecting special characters such as ‘\’, ‘”’, ‘;’, ‘<’ into user inputs. Typically, when a web program references user inputs in its HTML outputs, it expects that client-side browsers treat those inputs as only data. But, injected characters cause these browsers to interpret them as code. Therefore, an XSS exploit is generally achieved by illegally switching to a code context from a data context. OWASP [109] specified systematic XSS prevention rules to follow to ensure that any user input referenced in an HTML output is only treated as data. The rules condition that appropriate escaping mechanism be applied to user input according to the HTML context in which the input is referenced. Escaping disables the effect of special characters contained in user input and prevents them from invoking client side interpreters. Therefore, as long as user inputs are to be referenced in typical HTML contexts as data, XSS vulnerabilities can be completely avoided by following OWASP’s rules and there is also no harm in escaping the referenced tainted data even if the HTML output is not actually vulnerable.

Hence, our proposed approach is built on following the rules defined by OWASP to remove XSS vulnerabilities in server programs. We shall briefly review these rules in this section. Interested readers may refer to OWASP [109] for more detail.

- **Rule#0:** Do not reference user inputs in any other cases except the ones defined in Rule#1 to Rule#5. In some contexts, no special character is required to perform XSS injection, meaning that escaping rules could become complex or impossible to prevent an exploit. Thus, escaping rules in Rule#1 to Rule#5 only apply to the typical contexts where user inputs are commonly referenced. This Rule#0 conditions that no user input is to be directly referenced in any other cases. This rule is the most important among all XSS prevention rules because it implies a whitelist approach, that is, it accepts only known good cases and rejects all other cases (by contrast, a blacklist approach rejects only known bad cases but accepts all other cases. Thus, blacklist approach is commonly known as weak and insufficient).
- **Rule#1:** Use *HTML entity escaping* for the tainted data referenced in an *HTML element*. For example, `<body><div>htmlEscape(tainted_data)</div></body>`, where “`htmlEscape()`” is the HTML entity escaping method, conforms to this rule.
- **Rule#2:** Use *HTML attribute escaping* for the tainted data referenced as a *value* of a *typical HTML attribute* such as `name` and `value`. This rule does not apply to the two dangerous attributes—`href` and `src`. The only acceptable way to reference tainted data as values of `href` and `src` attributes is stated in Rule#5. Any other cases of tainted data referenced in the contexts of `href` and `src` are disallowed under OWASP’s escaping rules and OWASP recommends the use of only programmer-defined data in such cases because the referenced tainted data may simply point to a JavaScript source without the use of special characters. This rule also does not apply to all event-handler attributes such as `onclick`. Event-handler attributes should be handled according to Rule#3. For example, `<input value='htmlAttrEscape(tainted_data)'\>`, where “`htmlAttrEscape()`” is the HTML attribute escaping method, conforms to this rule; however, `<a href='htmlAttrEscape(tainted_data)'\>` does not.
- **Rule#3:** Use *JavaScript escaping* for the tainted data referenced as a *quoted data value* in a *JavaScript block* or an *event-handler*. This rule does not apply to the tainted data referenced as any other ways in a code block except as a quoted data value. For example, `<body onload="x='javascriptEscape(tainted_data)'">`, where “`javascriptEscape()`” is the JavaScript escaping method, conforms to this rule. In this thesis, we only discuss XSS injection using JavaScript. However, this rule applies to other client side scripts such as VBScript and Flash.

- **Rule#4:** Use *CSS escaping* for the tainted data referenced as a *value* of a *property* in a *CSS style*. For example, `<table style="width:cssEscape(tainted_data)">`, where “`cssEscape()`” is the CSS escaping method, conforms to this rule.
- **Rule#5:** Use *URL escaping* for the tainted data referenced as a *HTTP GET parameter value* in a *URL*. For example, `<a href='http://www.site.com?name=urlEscape(tainted_data)'\>` and `<img src='http://www.site.com?imgid=urlEscape(tainted_data)'\>`, where “`urlEscape()`” is the URL escaping method, conform to this rule.

Escaping APIs: OWASP [110] provides enterprise security APIs (ESAPI) that can be used to enforce the above XSS prevention rules in vulnerable web applications. As these APIs are used in our proposed approach and our implementation, we shall briefly review the ESAPI project in the following.

ESAPI is a security project that facilitates users to enforce security in both developed and developing web applications. The project is implemented for different web languages such as Java, .NET, and PHP. It provides a variety of security mechanisms such as authentication, validation, encoding, encryption, security wrappers, filters, and access control to mitigate various web security issues. As such, with ESAPI, users have a choice of implementing any of these mechanisms in their applications.

To utilize the security controls provided by ESAPI, users must first install ESAPI project into their applications. The documentation on ESAPI installation and configuration procedures can be found in ESAPI package downloadable from `<code.google.com/p/owasp-esapi-java/>`. The Java documentation on ESAPI’s escaping/encoding APIs can be found in

`(owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Encoder.html)`

Once ESAPI is installed, a user could secure the tainted data by wrapping it with a proper escaping API before it is referenced in an HTML output. The proper API is to be determined based on the corresponding HTML context and the above XSS prevention rules. For example, if the context is *HTML element*, Rule#1 applies and thus ESAPI’s *HTML escaping* API is to be used as shown in the following:

```
<div>ESAPI.encoder().encodeForHTML(tainted_data)</div>
```

It ensures that the tainted data is unable to cause context switching from its residing HTML element context. The appropriate APIs for the other rules—Rule#2 to Rule#5 are “`encodeForHTMLAttribute()`”, “`encodeForJavaScript()`”, “`encodeForCSS()`”, and “`encodeForURL()`”, respectively.

6.2 Proposed Approach

Our proposed approach consists of two major phases—detection and removal. The first phase identifies the potential XSS vulnerabilities in a given server program. The second phase first identifies the code locations where the tainted data can be adequately escaped and determines the required escaping mechanisms, and then escapes the tainted data using ESAPI's APIs. Our approach strictly follows the XSS prevention rules discussed in Section 6.1. That is, it applies ESAPI's escaping mechanisms if and only if the case belongs to Rule#1-Rule#5. If the case belongs to Rule#0 (any other cases not belonging to Rule#1-Rule#5), the proposed algorithm provides two options to user in order to secure the concerned statement: (1) Lenient mode—it requests the user to provide an appropriate sanitization method; (2) Strict mode—it unconditionally removes the tainted data from the code location it is referenced. Therefore, our XSS vulnerability removal algorithm is sound and complete in terms of removing all the XSS vulnerabilities in server programs. In the following, we provide the details of the approach.

6.2.1 Detection

This phase is based on the taint-based analysis technique adapted from our work on vulnerability auditing presented in Chapter 5. In the following, we briefly review the work and present the method for extracting XSS vulnerability information from a server program.

We adopt the basic definitions of the control flow graph (CFG), such as control and data dependence, defined by Sinha et al. [140]. In addition, we shall adopt the following definitions.

In a CFG, a node x is *transitively data dependent* on a node y if there exists a sequence of nodes, $y_0 = y, y_1, y_2, \dots, y_n = x$, in the control flow graph such that $n \geq 2$ and y_j is data dependent on y_{j-1} for all $j, 1 \leq j \leq n$. The *input node* is a node i at which the data that may be controlled by an external user is accessed. Thus, input nodes include all the nodes which access tainted data from direct input sources, such as HTTP request parameters, HTTP headers, and cookies; and indirect input sources, such as session variables, persistent objects, and database records. Direct input sources are known as sources of reflected XSS. Indirect input sources are known as sources of stored XSS.

A node o is called an *HTML output node* if o produces an HTML response output. We define the HTML output node o as a *potentially vulnerable output node (pv-out)* if o satisfies at least one of the following conditions:

- 1) o is also an input node (e.g., `out.print(req.getParameter("input"))`).
- 2) o is data dependent on an input node i .
- 3) o is transitively data dependent on an input node i .

Based on the above definitions, in our work on vulnerability auditing presented in Chapter 5, we have implemented the identification of pv-outs by tracking the flow of tainted data between input nodes and HTML output nodes. We make use of this work to extract the potential XSS vulnerability information from a given program and supply the extracted information to the next phase.

As an illustration, in Figure 6-1a, statement 1 and 2 define the variables `memID` and `pwd` respectively with the data from a direct input source— HTTP request GET. Statement 5 defines the variable `name` with the data from an indirect input source— Database. Statement 11 references the tainted data through the variable `html`. The CFG shown in Figure 6-1b represents the program in Figure 6-1a. Nodes 1, 2, and 5 in Figure 6-1b are input nodes. Node 11 is a pv-out because it satisfies the third condition of the above definition of pv-out.

```

public class Login extends HttpServlet {
    public void doGet (HttpServletRequest req,
        HttpServletResponse resp){

1   String memID = req.getParameter("id");
2   String pwd = req.getParameter("password");
3   String html = "<HTML><BODY><h1>"; //HTML structure
    . . . //perform database access & input validation
4   if(LoginValid) {
        //retrieve Member Name from database
5       String name = rs.getString("LastName");
6       html += "Welcome " + name + "! </h1>";
    }
    else {
7       memID = "S123456";
8       html += "Welcome Guest! </h1>";
    }

9   memID = memID.substring(0,7);

10  html += "<input type='hidden' name='member_id'
        value= '"+memID+"'>";
11  out.print(html);
    . . .
}

```

Figure 6-1. (a) Code snippet of a sample Login Servlet program. (b) CFG of the program.

6.2.2 Removal

This phase contains two major steps— HTML context discovery and secure source code replacement. The first step first identifies the statements at which the tainted data referenced in an HTML output statement can be escaped without compromising intended HTML outputs and security aspects. Then it extracts the HTML document structure surrounding every tainted data from the source code and identifies the HTML context using pattern matching. The required escaping mechanism for every tainted data is then determined based on the context identified and the XSS prevention rules (Section 6.1). The second step generates secure code structures using OWASP’s escaping APIs [110] as replacements for original code structures.

The algorithms are mainly based on data flow analysis and pattern matching. Therefore, static program analysis tools such as *Soot* [142] can be used to implement them. Next, we shall provide the detail of the two algorithms.

HTML context discovery: When an HTML output statement is identified as vulnerable, the tainted data referenced in that statement must be escaped according to the HTML context the referenced data is in. However, in some scenarios, escaping should not be done in the vulnerable statement itself because the variable containing the tainted data may also contain programmer-defined HTML document structures. Escaping may not be done in the input statements as well because the variables defined in the input statements may involve in more than one HTML context depending on different program paths. Therefore, for each pv-out, the algorithm first finds the statements at which the tainted data can be properly escaped (we shall address such statements as *escaping statements* and the nodes representing them in a CFG as *escaping nodes*). Then, the algorithm identifies the HTML context by analyzing the HTML document structure surrounding the escaping statements. This technique is further explained in the following:

Let o be a pv-out in the CFG of a program. The following three conditions ensure that there is always an escaping statement for every pv-out.

- If o satisfies the first condition of the definition of pv-out, the algorithm marks o as *escape_stmt* and also marks the method that retrieves the tainted data (e.g., `req.getParameter("input")`) as *to_be_escaped*.
- If o satisfies the second condition of the definition of pv-out, there is an input node i and a variable v defined in i and used in o . The algorithm marks o as *escape_stmt* and also marks v as *to_be_escaped*.
- If o satisfies the third condition of the definition of pv-out, there is at least one sequence of nodes, $\{i = x_0, x_1, \dots, x_n = o\}$, such that o is transitively data dependent on an input node i (note that there may be more than one sequence of nodes because tainted data may flow from input node(s) to pv-out through different program paths). For each sequence of nodes, the algorithm tracks the flow of tainted data from the input node i to the node x_j , $1 \leq j \leq n$; while tracking, if a node x_j performs a string operation which concatenates that tainted data with another variable or a raw string data, the algorithm (1) computes the nodes on which x_j is (transitively) data dependent and extracts any raw string data found in those nodes; and (2) marks the node x_j as *escape_stmt* and the tainted data (i.e., the variable containing the tainted data) referenced in x_j as *to_be_escaped*, if the extracted data contains any HTML special character such as '<'. If there is no such x_j exists, the algorithm simply marks o as *escape_stmt* and also marks the tainted data referenced in o as *to_be_escaped*. This is

because there is no node from i to x_n that could integrate any legal or programmer-defined HTML document structure into the tainted data.

After an escaping statement is identified for a pv-out, the algorithm examines any raw string data used in the escaping node and applies pattern matching to extract any identifiers found in those data that can be identified as HTML document structure. For pattern matching, the algorithm uses an HTML pattern library that stores the patterns of HTML document structures. The document patterns are defined according to HTML 4.01 specification from W3C recommendation [154] (note: as XHTML 1.0 [155] and HTML 4.01 are basically the same language in terms of the definitions of elements and attributes addressed in Rule#1-Rule#5, our defined patterns could also be used for matching scripts written in XHTML). The algorithm matches any extracted HTML document structure against those patterns from the library and attempts to identify the corresponding HTML context. If the context cannot be identified, the algorithm continues to explore (1) the nodes on which the escaping node is (transitively) data dependent and (2) the HTML output nodes surrounding the escaping node and the nodes on which those output nodes are (transitively) data dependent; and analyze the raw string data found in those nodes in order. Once the context is identified for the tainted data referenced in the escaping node, an appropriate escaping mechanism is determined based on the XSS prevention rules discussed in Section 6.1. For example, if the HTML context is identified as *HTML element*, according to Rule#1, *HTML entity escaping* is required. If the algorithm cannot identify the HTML context until a preset timeout or there is no further node to explore, it assumes that the case belongs to Rule#0. Therefore, this algorithm ensures that any tainted data referenced in an HTML output statement shall be secured by applying one of the XSS prevention rules described in Section 6.1.

Note that there may be more than one escaping statement for a pv-out because (1) it may satisfy more than one condition of the definition of pv-out and (2) there may be more than one input node influencing the pv-out. The above pattern matching procedure is to be performed for every escaping statement.

As an illustration, Figure 6-2 shows the pattern matching analysis performed for the pv-out in the Login Servlet program from Figure 6-1. The sequences of nodes {1, 9, 10, 11} and {5, 6, 10, 11} shown in shaded color in Figure 6-2 are the two sequences of nodes on which the pv-out, node 11, is transitively data dependent, and node 1 and 5 are the input nodes. According to the above HTML context discovery algorithm, for each sequence of nodes, the algorithm traverses the nodes starting from the input node and extracts any identifiers found.

Figure 6-2. Sequences of nodes that are analyzed and HTML document structures collected from string objects while identifying the HTML contexts of tainted data referenced in a vulnerable node. Both CFGs represent the Login Servlet program from Figure 6-1.

As shown in Figure 6-2a, for the nodes {1, 9, 10, 11}, the HTML document structure after analyzing node 10 is “<input type='hidden' name='member_id' value= ' "+memID+" '>”. The algorithm marks node 10 as *escape_stmt* and the variable *memID* as *to_be_escaped* as the algorithm tracks that the tainted data accessed in node 1 has been assigned to *memID*. The document pattern extracted at node 10 is then matched against the patterns from the library and it is identified as *HTML attribute context* which corresponds to Rule#2. Similarly, as shown in Figure 6-2b, for the nodes {5, 6, 10, 11}, when the algorithm traverses node 6, it identifies some HTML special characters; thus it marks node 6 as *escape_stmt* and the variable *name* as *to_be_escaped*. During pattern matching step, after analyzing the escaping node, node 6, the algorithm continues to analyze the nodes on which the escaping node is (transitively) data dependent. Therefore, node 3 is analyzed next. From node 6 and 3, the document structure extracted is “<HTML><BODY><h1>"+ name + “! </h1>”. It is then identified as *HTML element context* which corresponds to Rule#1.

Secure source code replacement: As an input, from the previous step, this algorithm receives the information of *escape_stmts*, *to_be_escapeds*, and *to_be_escapeds'* associated escaping rules and required escaping methods. Firstly, it declares the required ESAPI packages [110] on top of

the program under test. Next, for each *escape_stmt* in each pv-out *o*, the algorithm wraps the tainted data referenced in *escape_stmt* with appropriate escaping APIs in the following three steps:

- 1) Identify the appropriate escaping API *escape_api* from ESAPI library that corresponds to the required escaping mechanism of the variable or the method marked as *to_be_escaped* in *escape_stmt*. For example, if the required escaping mechanism is *HTML entity escaping*, the appropriate API to use is `ESAPI.encoder().encodeForHTML()`. The corresponding APIs for other escaping mechanisms are discussed in Section 6.1.
- 2) Modify *escape_stmt* by wrapping the object marked as *to_be_escaped* with *escape_api*.
- 3) Delete or comment (for program maintenance purposes) the original statement and insert the above modified statement into the same code location.

If the pv-out corresponds to Rule#0 (i.e., either the identified HTML context does not belong to the contexts stated in Rule#1-Rule#5 or no context could be identified), the algorithm provides two options to user. If the lenient option is chosen, the algorithm reports the corresponding XSS vulnerability information to user, requests the appropriate sanitization/escaping scheme, and sets the user's input as *escape_api*. If the strict option is chosen, the algorithm sets a default sanitization method which returns an empty string as *escape_api*. Secure source code replacement is then performed in the same procedure as the above steps 2 and 3 using this escaping API *escape_api*. For example, in the following JSP code, the tainted data is referenced in a complex HTML context which does not belong to any of the contexts stated in Rule#1-Rule#5:

```
<img src= '<%=tainted_variable%>' />
```

HTML attribute “src” is considered complex because tainted data referenced as a value for this attribute could easily invoke (have many different ways to invoke) a malicious script code without the use of special characters or keywords (e.g., `<img src= 'http://hackersite/hack.js' />`). For such a case, our algorithm in strict mode performs the following:

```
<img src= '<%=saferXSS.emptyStrAPI(tainted_variable)%>' />
```

The resulting HTML output would always be:

```
<img src= '' />
```

Therefore, our algorithm in strict mode shall produce unintended HTML outputs for the cases belonging to Rule#0. However, as we have discussed, such cases always involve high security risks and no escaping or sanitization is often possible to avoid the risks.

<pre> import org.owasp.esapi.ESAPI; public class Login extends HttpServlet { . . . String memID = req.getParameter("id"); String pwd = req.getParameter("password"); String html = "<HTML><BODY><h1>"; if(LoginValid) { String name = rs.getString("LastName"); //html += "Welcome " + name + "! </h1>"; html += "Welcome " + ESAPI.encoder().encodeForHTML(name) + "! </h1>"; } else { memID = "S123"; html += "Welcome Guest! </h1>"; } memID = memID.substring(0,7); //html += "<input type='hidden' name='member_id' value= '\"+memID+'\">"; html += "<input type='hidden' name='member_id' value='\" + ESAPI.encoder().encodeForHTMLAttribute(memID)+ \"'>"; out.print(html); </pre>																													
<p><u>Final Report</u></p> <p><i>Login Servlet</i></p> <table> <tr> <th><i>pv-out</i></th><th><i>input</i></th><th><i>escaped data</i></th><th><i>escaping mech</i></th><th><i>escaping stmt</i></th><th><i>comment</i></th></tr> <tr> <td>Line 11</td><td>Line 1</td><td>memID</td><td>HTML attribute</td><td>Line 10</td><td>modified</td></tr> <tr> <td>Line 11</td><td>Line 5</td><td>name</td><td>HTML entity</td><td>Line 6</td><td>modified</td></tr> <tr> <td colspan="6">. . .</td></tr> </table>						<i>pv-out</i>	<i>input</i>	<i>escaped data</i>	<i>escaping mech</i>	<i>escaping stmt</i>	<i>comment</i>	Line 11	Line 1	memID	HTML attribute	Line 10	modified	Line 11	Line 5	name	HTML entity	Line 6	modified	. . .					
<i>pv-out</i>	<i>input</i>	<i>escaped data</i>	<i>escaping mech</i>	<i>escaping stmt</i>	<i>comment</i>																								
Line 11	Line 1	memID	HTML attribute	Line 10	modified																								
Line 11	Line 5	name	HTML entity	Line 6	modified																								
. . .																													

Figure 6-3. Login Servlet program secured with OWASP’s security APIs [110] and the report produced by the algorithm.

Hence, removal of all XSS vulnerabilities from input programs can be fully automated using the above algorithms. And code modification required is very minimal because only objects containing tainted data are wrapped with escaping API calls. To facilitate software maintenance and security audit, the algorithm also reports (1) the source line numbers of modified statements, input statements, and pv-outs; (2) the escaped data; and (3) the associated escaping mechanisms used. For the Login Servlet program in Figure 6-1, the algorithm will produce the output as shown in Figure 6-3 (for brevity, only the necessary statements are shown). The modified statements are shown in bold.

6.3 Evaluation

We developed a prototype tool called *saferXSS* to implement the proposed approach. Using the tool, we evaluated the proposed approach on five open-source web applications. In the evaluation, we sought to answer the following two research questions:

RQ1. Does the approach compromise the required HTML outputs of the programs?

RQ2. Is the approach effective in removing the XSS vulnerabilities found in real-world applications?

In the following, we discuss the implementation and the evaluation of our proposed approach in detail.

6.3.1 Prototype Tool

The prototype tool, *saferXSS*, was developed through the use of program analysis tool called *Soot* [142]. Figure 6-4 shows the architecture of *saferXSS*. The tool consists of three modules; Program Analyzer, XSSV Detector, and XSSV Remover. It receives Java Servlet files as inputs. For each input program, *Program Analyzer* uses *Soot*'s APIs to build the CFG and stores the properties of each control flow node in global variables. *XSSV Detector* includes two major modules: data tracer and identifier. The two modules combine together to implement the XSS vulnerability detection phase discussed in Section 6.2.1. *Data tracer* traverses each node in the CFG and finds the HTML output nodes which reference tainted data. Then, for each node found, *identifier* performs data dependency analysis to determine the nodes where escaping is to be performed. *XSSV Remover* consists of two major modules: context finder and code wrapper. These two modules implement the two algorithms discussed in Section 6.2.2. *Code wrapper* provides a user interface for user to set lenient or strict mode. In lenient mode, whenever a vulnerable statement belonging to Rule#0 is encountered, the interface shows the vulnerability information to user and requests the appropriate sanitization/escaping method from user. As outputs, the tool produces modified Java Servlet programs and vulnerability report containing information of the potential XSS vulnerabilities found in the program. As a final step, user has to install OWASP's ESAPI into the application and re-compile the modified programs.

Figure 6-4. Architecture of the prototype tool, *saferXSS*

6.3.2 Test Subjects

For evaluation, we selected five Java-based open source applications, *Events*, *Classifieds*, *Roomba*, *PersonalBlog*, and *JGossip*, as test subjects. *Events* and *Classifieds* are downloaded from GotoCode [44]. *Roomba*, *PersonalBlog*, and *JGossip* are downloaded from SourceForge [143]. *Events* (3,818 LOC) is an event management system where users may participate or organize events. *Classifieds* (5,745 LOC) is an online classifieds system where users may advertise various items or visit the system for online shopping. *Roomba* (3,438 LOC) is a room booking system for small to medium-sized hotels. *PersonalBlog* (17,149 LOC) is an online blogging system where various users can publish their contents online. *JGossip* (79,685 LOC) is a forum system for discussion of various topics among users and it is a large-scale application. *Events* and *Classifieds* have been used as benchmarks in related works [48, 49]. *Roomba* has been used in [79]. *PersonalBlog* and *JGossip* have been used in [85]. The LOC counts do not include library classes. Inputs from anonymous users are common in these applications. Therefore, XSS vulnerability is a serious security concern since a malicious user may easily trick many innocent users through exploiting an XSS hole.

Considering that our test subjects' sizes vary from small to large scales and that the test subjects have different functionalities and characteristics, they are good representatives of many open source web applications in the wild. But, being open source, our test subjects may not be representatives of industrial applications. To be a credible empirical study, the software engineering community demands that the test subjects meet the following criteria [153]: (1) developed by a group of professionals, rather than novices or students individually; and (2) developed under industrial settings, rather than arbitrary or artificial settings. Some of our test subjects like *PersonalBlog* and *JGossip* meet these requirements, and they are widely-used, real world applications. Furthermore, all the above subjects have also been used in evaluating approaches such as [48, 49, 79, 85], which are closely related to our work. We, therefore, believe that our selected test subjects are good samples for the evaluation of our approach, and that choosing a different set of test subjects including industrial applications for evaluation would not vary the outcome much.

6.3.3 Experiment

We conducted experiments on each test subject by performing the following procedure. First, we specified the directory of the subject's source folder for *saferXSS*, set the tool in strict mode, and ran it. The tool analyzed all the Java files and modified the programs with potential XSS vulnerabilities. It also produced the vulnerability report which contains information about modified program statements (such as the one shown in Figure 6-3). Independently, we also inspected the source code of each subject to confirm the potential vulnerabilities reported by the tool. Table 6-1 shows the results of the tasks performed by the tool. Each reported vulnerability corresponds to one program statement (i.e., a pv-out). Table 6-2 shows the statistics obtained from the tool and our code inspection result. It shows the types of HTML contexts (observed in the test subjects) in which tainted data is referenced, and the numbers of the potential XSS vulnerabilities (reported by the tool) and the actual ones among the potentials (inspected by us), which correspond to each HTML context type. We observed that the actual XSS vulnerabilities present in the test subjects are mainly caused by improper input sanitization (*Classifieds*, *Events*), total absence of sanitization (*Roomba*), or failure to identify all input sources (*PersonalBlog*, *JGossip*). Note that the numbers of potential XSS vulnerabilities in Table 6-2 include both actual ones and false positives.

Table 6-1. XSS Vulnerabilities removed by *saferXSS* tool

Test Subject	#Stmt potentially vuln. to XSS	#Modified Stmts	#Modified Servlets
Events	81	81	11
Classifieds	123	123	10
Roomba	153	153	28
PersonalBlog	84	84	14
JGossip	312	312	40

Table 6-2. Actual XSS vulnerabilities found in the test subjects and potential XSS vulnerabilities reported by *saferXSS* tool, which correspond to each HTML context type

HTML context	#Stmt vuln. to XSS / #Stmt potentially vuln. to XSS				
	Events	Classifieds	Roomba	PersonalBlog	JGossip
HTML element	0/0	0/85	62/78	0/35	12/85
HTML attribute	9/28	9/17	42/45	0/12	12/37
JavaScript	0/0	0/0	2/2	0/0	0/3
CSS	0/0	0/0	0/0	0/0	0/83
URL	11/53	10/21	18/23	1/11	4/101
Rule#0	0/0	0/0	5/5	0/26	0/3
Total	20/81	19/123	129/153	1/84	28/312

Second, we installed ESAPI project into the modified subject, set up both the original and modified subjects on Eclipse [31], and deployed them on Tomcat 6.0 server⁸. We also configured the database connections with MySQL 5.0⁹ and ran the subject's SQL files on MySQL database to generate the required tables.

Third, based on the XSS vulnerability report produced by the tool, we manually created a test suite, called functional test, in which a test case was formed for each modified program statement. This test suite was designed to test if the modified statements still produce the HTML outputs as intended. We then manually executed the test cases on both the original and modified subjects and compared the results (i.e., the resulting HTML documents). If the modified subject produces the same output as the original subject for a given test case, it is counted as *one test passed* (we did not

⁸ <http://tomcat.apache.org>

⁹ <http://www.mysql.com>

consider the actual functional requirements of the subject). Otherwise, it is counted as *one test failed*.

Fourth, based on the actual XSS vulnerability information inspected by us, we created another test suite, called XSS test, in which a set of test cases was formed for each actual XSSV observed in the test subject. Each set of test case was constructed with a variety of XSS attack vectors designed to exploit each specific XSSV. The attack vectors were crafted based on the HTML context and escaping statement information reported by the tool. The attack patterns were also referenced from RSnake [119] and Kiežun et al.'s attack library [68]. They were crafted in such a way that a successful attack results in an alert message or a malformed web page. If no attack from a given set of test cases succeeds, it is counted as *one test passed*. If one of the attacks from the set succeeds, it is counted as *one test failed*. We again manually executed the test cases on both the original and modified subjects, and verified the results.

As an illustration, some XSS attack patterns and the results of their executions on both the original and modified codes are shown in Table 6-3. The web pages shown in the first column are the places the attacks occurred. The attack vectors were injected into the input parameters shown in the third column. The injection may not take place in the same web page where the attack occurred depending on the stored and reflected XSS scenarios. For example, to expose the vulnerability in the web page 'bookings.step3' of *Roomba*, the attack vector ' `; alert('XSSed');`' was injected into the HTTP request GET parameter `firstname` in the web page 'bookings.step1' where the data was then stored into the database.

Finally, we recorded the results of functional and XSS test cases executed on both original and modified test subjects. The results are listed in Table 6-4.

Table 6-3. Example of XSS attacks performed on original and modified test subjects

Web Page	HTML Context	Attack Vector	Original Code	Result	Modified Code	Result
Roomba /viewTable	Rule#0 (URL inside JavaScript)	chosenTable = ' ' ; ! -- "<XSS>=	<script> var newUrl = "viewTable.jsp?tableId="+<%=chosenTable %> </script>	Mal- formed web page	<script> var newUrl = "viewTable.jsp?tableId="+<%=saferXSS.emptyStringAPI(chosenTable) %> </script>	No mal- formed web page
JGossip/ jspf. message Form	HTML element	name = <SCRIPT> alert('X SSed')</ SCRIPT>	<td><%= request.getParameter("name") %></td>	Alert message "XSSed"	<td><%= ESAPI.encoder().encodeForHTML(request.getParameter("name")) %></td>	No alert message
Events/ Login	HTML attribute	querystring=' ' ; ! -- "<XSS>=	<input type='hidden' name='querystring' value=<%=getParam(request,"querystring") %>>	Mal- formed web page	<input type='hidden' name='querystring' value=' '+ESAPI.encoder().encodeForHTMLAttribute(getParam(request,"querystring")) %>>	No mal- formed web page
Roomba /bookings.step3	Java- Script	firstname = ' alert('X SSed');	<script> var firstname =<%=rs.getString("firstname") %>;</script>	Alert message "XSSed"	<script> var firstname =<%=ESAPI.encoder().encodeForJavaScript(rs.getString("firstname")) %>;</script>	No alert message
Personal Blog/deletepost	URL	id = ";<SCRIPT> alert('X SSed')</ SCRIPT>	<a href="deletePost.do?method=executeFinish&postId=<%=id %>">	Alert message "XSSed"	<a href="deletePost.do?method=executeFinish&postId=<%=ESAPI.encoder().encodeForURL(id) %>">	No alert message

6.3.4 Result and Discussion

Functional integrity: As shown in Table 6-1 and Table 6-2, the *saferXSS* tool produced false positives for all the test subjects (#False positives = #Stmt potentially vuln. to XSS – #Stmt vuln. to XSS). False positives are expected because our proposed approach tends to be conservative. Remember that the main objective of our approach is to achieve absolute security by securing all the “potentially vulnerable” data reported. False positive arises when the secured data is not actually vulnerable. For example, in our approach, data accessed from database is considered as tainted, and thus, it is escaped and secured. But the data may actually be clean if it is written into

the database by the programmer himself or if it is the auto-increment value generated by the database. Our manual inspections confirm that the test programs do access data from user sessions or database defined by the programmer or the database (e.g., auto-increment primary key). Such data is not exploitable by attackers and thus, these are false positive cases. However, our secure code replacement procedure is performed such that it does not affect the resulting HTML outputs or other program operations (discussed in Section 6.2.2), except for cases that belong to Rule#0, which are discussed in the following.

As shown in Table 6-2, 5 cases from *Roomba*, 26 cases from *PersonalBlog*, and 3 cases from *JGossip* correspond to Rule#0. From our code inspection, we observed that all the five cases from *Roomba* are actually vulnerable whereas all the cases from *PersonalBlog* and *JGossip* are false positives. Observing that *PersonalBlog* has many false positives with respect to Rule#0, we again manually inspected those specific cases and found that *PersonalBlog* is mostly written in complex HTML document structures with accesses to user input data. For example, we observed the use of user input together with ‘innerHTML’ property in JavaScript. Securing such input is not straightforward since there are many exploitation possibilities in such scenario. Our approach considers them as Rule#0 cases (i.e., cases that proper security procedure cannot be determined).

Nevertheless, all those 34 cases were secured with our tool’s strict mode-sanitization method which returns an empty string. Therefore, as shown in Table 6-4, the functional tests corresponding to those cases failed for the modified test subjects. In a real scenario, a user may also choose the lenient mode; and based on the vulnerability information provided by the tool, he may decide whether to sanitize the tainted data or to ignore the risk. For all other test cases, we confirmed that the resulting HTML documents of both the original and modified subjects were the same. Therefore, the results show that the proposed escaping procedure performed on the cases belonging to Rule#1-Rule#5 does not affect the intended HTML outputs of the programs though the escaping performed on the cases belonging to Rule#0 might affect the intended output. This answers our first research question RQ1.

Effectiveness: As the numbers of modified statements in Table 6-1 show, the *saferXSS* tool secured all the potential vulnerabilities identified by the tool. From our manual inspection, the potential XSSVs reported by the tool include all the actual XSSVs found in the test subjects. As the XSS test results in Table 6-4 show, the attacks from XSS test suite successfully exploited all the actual XSSVs in all the original subjects. None of the attacks was successful against the modified subjects. These results confirm that the actual XSSVs inspected by us are really exploitable. Quantitatively, the *saferXSS* tool removed 100% (197/197) of the actual XSSVs in the

test subjects with the precision of 26.2% ($\#actual/\#potential = 197/753$). More importantly, this experiment demonstrates that the proposed approach is effective in completely removing all the real XSSVs, which answers our second research question RQ2.

Table 6-4. Results of functional and XSS test cases executed on the original and modified test subjects

Subject	Test suite	Original subject	Modified subject
Events	Functional	All passed	All passed
	XSS	20 tests failed	All passed
Classifieds	Functional	All passed	All passed
	XSS	19 tests failed	All passed
Roomba	Functional	All passed	5 tests failed
	XSS	129 tests failed	All passed
PersonalBlog	Functional	All passed	26 tests failed
	XSS	1 test failed	All passed
JGossip	Functional	All passed	3 tests failed
	XSS	28 tests failed	All passed

6.3.5 Limitation

Our XSSV removal approach applies OWASP's escaping APIs only when the HTML context corresponds to the contexts defined in Rule#1-Rule#5 (Section 6.1). Tainted data referenced in any other contexts are unconditionally removed by the tool when set as strict mode.

Also, this approach does not track information flow across web pages. Thus, the tool loses precision when a tainted variable is first stored in a data store such as session objects and later referenced in an HTML output statement in a different web page. However, in order to prevent stored XSS, our proposed approach treats data accessed from such indirect input sources as tainted and then indiscriminately applies escaping procedures to all tainted data. As discussed in Section 6.2, doing so causes no harm as long as the input is to be referenced as a data in an HTML document.

Both our approach and tool target web applications written in Java language due to the frequent occurrences of XSS issues in Java-based web applications. However, the proposed idea can be easily extended to fit the syntax of other programming languages. Furthermore, our method modifies program source code instead of rewriting program bytecode. This is ineffective when the

application source code is not available. However, our approach not only performs vulnerability removal but also systematically reports the statements modified and the statements that have security risks (in lenient mode) so that the user can take further actions. As such, source code access is still required. It could though be future work to include this feature (bytecode rewriting) into our tool so that users could choose the preferred modification method.

6.4 Conclusion

In this chapter, we have presented a two-phase approach for finding and removing potential XSS vulnerabilities in server programs. The first phase adopts a taint-based analysis approach to track the flow of user inputs into HTML output statements and identify potentially vulnerable statements. The second phase uses pattern matching and data dependency analysis to identify the HTML contexts in which the user inputs are referenced and the required escaping mechanisms to guard against code injection. Then it performs source code generation and replacement to secure the potentially vulnerable statements using proper escaping APIs.

Some of the existing XSS mitigation techniques could detect many vulnerabilities or prevent most XSS attacks; however they do not remove vulnerabilities. As more and more sophisticated attack patterns are discovered, vulnerabilities if not removed could be exploited anytime. Our proposed approach focuses on removing vulnerabilities with minimal user intervention. We also developed the *saferXSS* tool that automates the proposed approach. In our evaluation, the tool was successful in removing all the XSS vulnerabilities found in five test subjects.

Chapter 7

RELATED WORK

Existing methods on mitigating SQLI and XSS threats can be broadly classified into five types—defensive coding practices, vulnerability prediction techniques, vulnerability detection techniques, vulnerability testing techniques, and runtime attack prevention techniques.

In this thesis, we have presented three types of techniques—vulnerability prediction, vulnerability auditing, and vulnerability removal that also mitigate SQLI and XSS threats. Hence, our works are related to the works from the above five categories. This chapter discusses and contrasts our proposed techniques with these related works. As we discussed in our surveys on SQLI and XSS mitigating approaches [130, 135], we believe that SQLI and XSS threats have to be defended from a combination or an overlapped use of different types of techniques at different stages of a software development lifecycle. Hence, our approaches are proposed not to replace these existing works but to improve or complement them at various software development stages. Our ideal SQLI and XSS defense system is depicted in Figure 7-1.

Figure 7-1. Overview of an ideal use of SQLI and XSS mitigation techniques during a software development process

© 2013 IEEE

7.1 Defensive Coding Practices

Since SQLI and XSS vulnerabilities arise from improper handling of inputs, defensive coding practices [25, 28, 36, 97, 107, 109, 111] that enforce guards against inputs are the best solution for defeating SQLI and XSS threats. Naturally, these practices are to adopt during implementation stage and often, developers are responsible for this task. The following summarizes the proper defensive coding methods suggested to developers in the literature.

Parameterized queries or stored procedures: Anley [6] and OWASP [107] showed that the use of properly-coded parameterized queries or stored procedures is the perfect solution to defend against SQLI threats. If developers use any of these two query construction methods instead of dynamic query building method (explained in Chapter 2.1), SQL injection shall be non-existent these days. Proper coding style of parameterized queries or stored procedures enforces developers to define the structure of SQL code first before including parameters to the query. Because parameters are bound to the defined SQL structure, no injection of additional SQL code is possible.

Escaping: Due to the flexibility and complexity of web browsers and the difference in parsing technologies, parameterized method is not used to prevent against XSS attacks (unlike a DBMS which has only one SQL interpreter to parse a given query, a browser uses various script

interpreters such as JavaScript, HTML, Flash, etc. to parse a given HTML document). Instead, to defend against XSS threats, escaping of user inputs is the best solution. OWASP [109] proposed rules that define proper escaping schemes for inputs referenced in different HTML contexts (i.e., HTML body, HTML attribute, JavaScript, etc.).

Escaping is similarly effective against SQLI threats [107]. In the case that developer favors dynamic queries due to its flexibility (instead of parameterization), escaping all user-supplied parameters is the next best option. But, as insufficient or improper escaping practices are common, the recommendation for developers is (1) identify all input sources to realize the parameters that need escaping; (2) apply proper escaping procedures according to the context in which the input data is referenced; and (3) use standard escaping libraries instead of custom escaping methods.

Data type validation: In addition to escaping, data type validation is a useful and important defensive coding method [25, 36]. Validating whether an input is string or numeric could easily and quickly reject type-mismatched inputs. This could also simplify the escaping process because validated numeric inputs need no further cleansing action and could be safely used in SQL and HTML output statements.

Whitelist filtering: Developers often use *blacklist filtering* to reject known bad special characters such as \, ' , , , and < from the parameters to avoid SQLI and XSS [36]. However, the safer and recommended filtering approach is to accept only inputs that are known to be legitimate. This approach is suitable for well structured data such as (email) addresses, dates, zip codes, social security numbers, etc. Developers could keep a list of legitimate data patterns and accept only the input data which match them.

Although these recommended practices are the best way to defeat SQLI and XSS, their application is labor-intensive, error-prone, and difficult to be rigorous and complete. Furthermore, developers are often the ones to implement these practices but as some of them are not well-trained in security, the implementation is often inadequate.

To alleviate these problems in the context of SQLI, McClure and Krüger [88] developed SQL DOM – a set of classes that provide automated data type validation and escaping. Developers are required to provide SQL DOM with database schema and construct SQL statements using its APIs. This framework is especially useful when a developer needs to use dynamic query construction method (instead of parameterized queries) for greater flexibility. Similarly, Cook and Rai [26] also provided special libraries to generate safe query objects. In the context of XSS, Robertson and Vigna [117] proposed a web application framework that enforces strong typing of

HTML documents so that intended document structures and inputs referenced in the documents can be separated, and violations of intended document structures can be checked at runtime. Yip et al. also developed a framework called Resin [167] that provides developers with interfaces for generating code assertions that define security policies while writing application code; and the framework checks for conformance of the defined security policies during runtime. Resin framework is also applicable to SQLI.

Although defensive code implementation using these special libraries and development frameworks would be certainly helpful, the drawback is that developers are required to learn new query-development paradigm or new programming environments.

Our XSS vulnerability removal approach presented in Chapter 6 is closely related to Thomas et al.'s work [147], which removes SQLI vulnerabilities by generating parameterized queries. They find potentially vulnerable SQL statements (i.e., statements built dynamically) in the programs and replace them with parameterized SQL statements. However, their approach is not designed to remove XSS vulnerabilities. Unlike securing a SQL statement with a prepared statement, the technique of securing an HTML output statement with an escaping API is more complex as it has to take into consideration the HTML context and the appropriate API to use. Furthermore, Thomas et al.'s current tool uses pattern matching to deduce SQL structures. So it only works on SQL structures built with explicit strings. Incorporation of program analysis techniques is required to deduce SQL structures built with data objects or through function calls. By contrast, our work applies static data dependency analysis to track variables and extract HTML document structures.

In summary, defensive coding practices are important, and they act as the primary defense against SQLI and XSS. However, due to the above explained drawbacks, even when developers adopt defensive coding practices, SQLI and XSS flaws might still be present in programs. Furthermore, defensive coding is only applicable for new code (implementation stage). It would require rewriting for existing, developed code. Hence, our techniques would complement them at various software development stages. For example, at software testing stage, our proposed code auditing approach would be useful for verifying the adequacy of defensive code implementations.

7.2 Data Mining Techniques

Defect prediction: Data mining models used by our supervised vulnerability prediction techniques (Chapter 3 and Chapter 4) are similar to those used in many defect prediction studies [7, 10, 17, 75, 91, 92, 99, 100, 106, 141, 148, 149, 170]. In these studies, defect predictors are generally built from static code attributes [21]. Static code attributes are attributes that can be

measured statically from specification, design, implementation, etc. [32]. Some examples are software design measures [17], LOC counts, Halstead's attributes [52], and McCabe's code complexity attributes [87]. These studies typically use performance measures, such as recall, probability of false alarm, precision, accuracy, and area under the curve (AUC) of probability of false alarm versus probability of detection, to evaluate the predictors.

Some of these studies [7, 66, 75, 90, 91, 148, 150] compared the predictive performances of several classification algorithms, such as C4.5, neural networks, Naïve Bayes, support vector machine, random forest, and logistic regression, using common datasets available in public domains such as PROMISE [114].

Results from some major studies [10, 17, 75, 91, 99] can be summarized as defect predictors which produce *recall* > 70% and *probability of false alarm* < 25% are useful in practice. Tosun and Bener [148] also reported that up to 71% of the software engineering effort could be saved by using defect predictors instead of applying random code inspections. They have mainly showed that static code attributes are useful and efficient as they can be cheaply collected [91].

However, Fenton and Neil [33] and Shepperd and Ince [137] criticized the reliability of predictors built from attributes like size and code complexity attributes. Some other researchers concerned that static code attributes can only provide limited accuracy [7, 92, 93, 100] and that there is no universal set of attributes that works on every application domain [149, 170]. Lessmann et al. [75] also reported that new software attributes should be discovered to improve the performance of all types of classifiers.

Motivated by the above findings, Arisholm et al. [7] and Nagappan et al. [98, 100] focused on finding additional software attributes to enhance accuracy. They reported that process attributes (e.g., developer experience and code churn) in addition to static code attributes could significantly improve prediction accuracy. However, they also reported that despite the improvement, process attributes are difficult to measure, and measurements are often inconsistent. Menzies et al. [92] also showed that tuning the classifier according to a user-specific goal (e.g., finding the fewest modules that contain the most errors) improves the classifier's performance without the help of process or other type of attributes. They counter-argued that static code attributes are still one of the few measures that can be collected consistently across many systems. Zimmermann et al. [170] proposed new set of static code attributes and reported that network measures on dependency graphs work better than conventional static code attributes in predicting defects in Windows-based software.

Similar to these defect prediction studies, we proposed and applied static code attributes in building vulnerability predictors (Chapter 3). Like some of these studies, in addition to static attributes, we also proposed and used dynamic code attributes to improve predictive performance (Chapter 4). We also used the classifiers (such as neural networks) that were ranked among the best by these studies. And we applied the same performance measures, such as recall and precision, to evaluate our predictors.

In contrast to these defect prediction studies, our study in this thesis targets security vulnerabilities in web applications. We also defined and proposed new set of attributes targeted at predicting vulnerabilities based on static and dynamic program analysis.

Vulnerability prediction: Although we follow and apply data mining technologies used by defect prediction studies, there are also prediction approaches that target web application vulnerabilities like our approach. These approaches are more closely related to our work.

Shin et al. [138] used code complexity, code churn, and developer activity attributes to predict vulnerable programs. They achieved 80% recall and 25% false alarm rate. Their assumption was that, the more complex the code, the higher the chances of vulnerability. But from our observations, many of the vulnerabilities arise from simple code. That is, if a program does not employ any input validation and sanitization routines, the code would be simpler but nevertheless contain many vulnerabilities. Furthermore, Shin et al. [138] requires process attributes such as developer activity. As discussed above, process attributes can be very expensive to collect and data collected may not be consistent across projects [7, 92]. By contrast, our predictors presented in this thesis are built from only product attributes (i.e., code attributes) which could be collected more consistently [7, 92].

Walden et al. [156] investigated correlations between the security resource indicator (SRI) and the numbers of vulnerabilities in PHP web applications. SRI is derived from publicly available security information such as past vulnerabilities, secure development guidelines, and security implications regarding system configurations. Neuhaus et al. [101] also predicted vulnerabilities in software components from the past vulnerability information, and the imports and function calls attributes. Their work is based on the concept that components which contain imports and function calls that are similar to known vulnerable components are likely to be vulnerable as well. They achieved 45% recall and 70% precision.

Gegick et al. [42] used static analysis alerts (i.e., outputs from a static fault analysis tool), code churn, LOC counts to predict vulnerable software components. Like our approach, they also used

static program analysis to collect attributes. However, their static analysis only generates programming-fault alerts and manual audition is required to determine whether those alerts could be warnings of security vulnerabilities. This task requires a security expert and might be error-prone. By contrast, our attribute collection process is fully automated.

The major and general differences between these existing vulnerability prediction approaches and our vulnerability prediction approaches presented in Chapter 3 and Chapter 4 are (1) existing approaches predict security vulnerabilities in general whereas our approach focuses on SQLI and XSS vulnerabilities; (2) we propose a novel set of code attributes that reflect the code patterns of input sanitization and validation defense against SQLI and XSS; and (3) most importantly, our predictors identify vulnerable program statements whereas existing vulnerability predictors only identify vulnerable software components or program files. Therefore, their results cannot be directly compared with ours due to different granularities of vulnerability targets. Still, it is noteworthy that compared to their general vulnerability predictors, our specialized predictors achieved high detection rates and significantly low false alarm rates in predicting SQLI and XSS vulnerabilities (see Chapter 3 and Chapter 4).

Un-supervised learning: All the above existing prediction approaches discussed are built on classification algorithms. We explained in Chapter 2.3 that classification-based methods suffer from two major drawbacks—1) labeled instances are required for training, and 2) learned classifiers could only detect known bad patterns. Therefore, in addition to classification methods, un-supervised learning methods such as cluster analysis are also used in defect prediction studies such as Seliya and Khoshgoftaar [124] and Zhong et al. [168]. In these approaches, similar to classification-based methods, software modules are represented with attributes such as code complexity attributes. Distance functions such as Euclidean function are then used to compute the distance between attribute vectors and measure similarity among software modules. Clustering algorithms such as k -means are then used to group software modules based on their similarities. A software engineering expert is then required to label each group as either defective or non-defective.

Our techniques presented in Chapter 4 also rely on cluster analysis to predict vulnerabilities. But, the existing related works target defects in software modules whereas we target web application vulnerabilities. To the best of our knowledge, cluster analysis has not been used in vulnerability prediction. Furthermore, they require human experts to label clusters whereas our work automates this task based on some assumptions.

Cluster analysis is also widely used in solving a different web security issue called *network intrusion detection* or *anomaly detection* [23]. Hence, here, we also review some intrusion detection techniques as our related work.

Portnoy et al. [113] showed that cluster analysis could identify numerous network intrusions based on the two assumptions that (1) normal instances (legitimate network accesses) are much more frequent than anomalous instances (intrusions) and (2) anomalous instances have characteristics different from normal instances.

Katos [64] evaluated the performances of three different un-supervised learning schemes which are cluster, discriminant, and logit analysis. Thamaraiselvi et al. [146] combined both classification and clustering techniques to improve the accuracy of intrusion detectors. Bayesian classifier is used to detect known intrusion attacks learned from available labeled training data and genetic-based cluster analysis is used to detect unknown (unseen) attacks.

In our work presented in Chapter 4, the same assumptions as Portnoy et al.'s [113] have been used for detecting vulnerable clusters. But our cluster analysis is on static and dynamic code attributes extracted from program source code whereas cluster analysis in intrusion detection studies is on network access patterns extracted from network traffic.

7.3 Static Analysis-based Vulnerability Detection

This type of methods focuses on identifying vulnerabilities in server-side scripts using static taint analysis techniques.

Researchers formulated the problem of XSS and SQL injection as one of information flow integrity [12]. As such, it can be avoided by restricting tainted data (i.e., user inputs) from affecting un-tainted data (e.g., programmer-defined SQL query structure). To achieve this requirement, Livshits and Lam [82], Jovanovic et al. [61], and Xie and Aiken [165] applied prominent static analysis techniques, such as flow-(in)sensitive analysis, context-sensitive analysis, alias analysis, and interprocedural dependency analysis, to carry out the following tasks: (1) identify input sources and sinks that consume input data (i.e., SQL statements and HTML output statements) and (2) check whether or not every flow from a source to a sink is subject to any input validation and/or input sanitization routine. Livshits and Lam's work [82] is based on points-to analysis using binary decision diagrams. It requires users to specify vulnerability patterns in a program query language called PQL [86]. Xie and Aiken's work [165] is based on block and

function summary information obtained from symbolic execution. Jovanovic et al. [61] includes alias analysis to improve the accuracy.

Some of the vulnerability detection approaches such as Livshits and Lam [82] require user specifications to perform vulnerability analysis. Manual specifications are labor-intensive and it is hard to be complete and accurate. Therefore, Livshits et al. [81] proposed a method to automatically infer information flow specifications from program code using an interprocedural data dependency analysis. Their approach analyzes information flow paths in programs based on probabilistic reasoning and intuition in order to identify external input sources, sensitive sinks (e.g., SQL query statement), and sanitization functions as accurately as possible.

To specifically address SQLI, Wassermann et al. [157] applied static analysis to verify that dynamic query strings do not contain type errors (e.g., string type instead of numeric type).

The advantage of static taint analysis method is that it provides a quick detection of potential vulnerabilities in source code and is relatively easy to be implemented and adopted by security personnel. However, these approaches typically suffer from precision issues due to one or more of the following limitations: (1) semantics of input validation or sanitization routines are not precisely modeled; (2) input validation using predicates is not considered; (3) vulnerability patterns need to be specified; and (4) user intervention is often required to state the tainted-ness of external or library functions that inputs pass through. All these limitations could result in both false negatives and false positives.

To provide more precision, Wassermann and Su [158, 159] used a static string analysis technique adapted from Minamide's work on approximating dynamic web page outputs [95] to model the effects of input validation and sanitization routines. Their approach checks whether the string values returned from those routines still contain string values that are blacklisted as XSS threats [158] or whether the string values returned are syntactically confined in SQL queries [159]. If yes, the routines used are concluded as correctly implemented or vice versa. Fu et al. [39] used symbolic execution [69] and hybrid constraint solver (string solver and numeric solver) to identify user inputs through which SQLI attack strings can be injected.

The above enhancements provide more accuracy because the effects of string operations applied to inputs are analyzed and evaluated. However, the major weakness of string analysis is that it is difficult to model complex string operations such as string-numeric interaction and therefore, it may still result in false positives if conservative approximations for handling them are made. On the other hand, most of these techniques adopt blacklist comparison technique to filter

or detect known bad inputs. But if the available list of known bad characters is incomplete, it would also result in false negatives.

Our proposed approaches in this thesis are closely related to static analysis-based vulnerability detection approaches because (1) the approaches apply static analysis techniques; (2) the analysis is based on program source code; (3) the approaches are best applied before software deployment; and (4) the approaches do not require runtime checking mechanisms.

In this thesis, as we aim to address the shortcomings of existing static approaches, we propose approaches that may serve as either complementary or alternative solutions to these existing approaches. As discussed above, existing static approaches suffer from high false positive rates and could also produce false negatives mainly due to the difficulties in analyzing input validation and sanitization methods implemented in programs. By contrast, our vulnerability predictors presented in Chapter 3 and Chapter 4 are learned from the past vulnerability information and the code attributes that are designed to reflect the characteristics of input validation and sanitization methods implemented in the programs. Hence, our vulnerability predictors have the ability to infer the correctness of the implementations of input validation and sanitization methods based on our proposed attributes.

On the other hand, after potential vulnerabilities have been automatically identified by detection techniques, code inspection would be required to pinpoint the real vulnerabilities due to the possibility of false positives. However, existing static analysis approaches lack focus on assisting to code verification to facilitate the debugging of defects in defensive code. Users of these approaches might end up checking the whole piece of code to verify the reported vulnerabilities. Even if the vulnerability identification is accurate, more information is generally required to fix the vulnerability. That is, one needs to understand the weakness of defense features implemented or the requirement of features not implemented. On the other hand, no information is reported for the statements identified as not vulnerable. But such information is often necessary for auditing because there might also be false negative cases due to incomplete user specifications, inappropriate assumptions, or some limitations of the tools currently implemented. Hence, our code auditing techniques presented in Chapter 5 would assist this code verification task. Thereafter, actual vulnerabilities identified have to be removed from source code. Again, doing so manually would be labor-intensive and error-prone; hence, our automated vulnerability removal technique presented in Chapter 6 would be useful for this task.

7.4 Vulnerability Testing

Like static analysis-based vulnerability detection methods presented in the above sub-section, this type of methods also aims to detect vulnerabilities in programs. However, the approaches presented in the following are testing-based, and hence, dynamic program execution or dynamic program analysis techniques are mainly used. Some approaches also include static taint analysis techniques.

Automated attack vector generation: Balzarotti et al. [9] proposed a hybrid approach combining static taint analysis technique with dynamic testing technique. Their approach aims to test the adequacy of sanitization functions for defending against SQLI and XSS attacks. They use a static string analysis method similar to that of Wassermann and Su [158] to first identify the potentially faulty sanitization methods. Then they simulate the identified methods with a set of test inputs containing attack strings and check if any attack could reach the sinks.

Artzi et al. [8], Kiezun et al. [68], Ruse et al. [120], and Wassermann et al. [160] perform concolic (concrete+symbolic) execution to capture program path constraints, and use a constraint solver to generate test inputs that explore various program paths. Upon reaching the sinks, they execute the traversed path with two sets of test inputs—one set contains ordinary valid strings and another set contains attack strings obtained from RSnake’s attack libraries [118, 119], and check the differences between the resulting program behaviors. Lam et al. [73] used Livshits and Lam’s points-to analysis method [82] to track the flow of tainted data in the program. Using this information, the program is instrumented for model checking purposes. They implemented a model checker called QED [85] to simulate the instrumented program with inputs that are likely to lead to a match with user-specified vulnerability patterns. The effectiveness of this approach depends on the completeness of vulnerability specification provided by user.

The above automated test case generation frameworks typically rely on concolic execution engines such as jFuzz [58], Cute and jCute [125], and java PathFinder [152], and string constraint solvers such as Hampi [67].

The major advantages of this type of approaches over our proposed techniques are that (1) it creates concrete attack vectors and therefore, there are no false positives; and (2) it automates both test case generation and vulnerability analysis. But, it also has a few weaknesses as compared to our proposed techniques: (1) it requires dynamic execution and analysis frameworks which may be computationally expensive or difficult to be used; (2) its effectiveness depends on the capabilities of underlying string constraint solvers; (3) the attack string library (used to generate test cases)

may not be perfect due to everyday introduction of new attacks; and (4) model checking, symbolic execution, and concolic execution methods are known to suffer from state space explosion problem (i.e., number of paths to be explored might explode in the presence of many loops) [20]. These weaknesses might result in both false positives and false negatives. For example, if the attack string library is incomplete, the sample attack vectors generated during testing might be trapped by the program's sanitization routines. But some real-life attack vectors may succeed in circumventing those sanitization routines. Similarly, if state space explosion occurs, vulnerabilities that exist at deep state spaces may be missed. Among the recent empirical studies, Lam et al.'s tool [73] struggled at testing a Java-based Web application *JGossip* (80K lines of code) due to the possibilities of over 30 billion test cases. Kiežun et al.'s tool [68] only achieved 14% line coverage for a PHP-based Web application *phpBB* (35K lines of code).

There is a trend in recent researches that focus on characterizing bug patterns using pattern analysis and predicting unknown bugs using pattern mining and graph mining techniques [24]. This might ease the problem of incomplete attack patterns. Moreover, some researchers have proposed more effective string constraint solving techniques such as Sushi [40] while others have looked into hybrid techniques incorporating random test input generation methods [84] and artificial intelligence algorithms such as search based software engineering [5, 164], which might ease the problem of state space explosion. But, at present, there is no universal solution to solve the state space explosion problem and much work is still required to address the above problems associated with dynamic analysis.

By contrast, techniques mainly used by our approaches are static analysis and data mining techniques which are light-weight and relatively easy to be adopted. Although our approach presented in Chapter 4 requires dynamic analysis, the analysis is only at functional level without the need to reason about the paths in programs. Moreover, our data mining-based approaches also have the advantage of being able to process many data instances, and therefore, our approaches proposed in this thesis provide an alternative, cheaper, and efficient mode of finding many vulnerabilities.

In the following, we discuss conventional input validation, black-box, and white-box testing techniques which can be used to mitigate XSS and SQLI threats. As these techniques mainly focus on test case generation and lack focus on automated testing and vulnerability analysis, they clearly contrast with our proposed approaches.

Input validation testing: Input validation testing (IVT) of web application checks whether an application accepts inputs that conform to program requirements while rejecting inputs that do not.

Since an application is highly likely to be vulnerable if input validation is not practiced, IVT approaches could also uncover SQLI and XSS vulnerabilities. In literature, specification-based and code-based IVT approaches have been proposed.

Specification-based IVT methods [54, 55, 76, 77, 104] generate test cases with the aim of exercising various combinations of valid/invalid input conditions stated in specifications. Since input validation implemented in client-side scripts can be easily bypassed by attackers, Offutt et al. [104] proposed a bypass testing technique that specifically checks the adequacy of input validation implemented in server-side scripts. To avoid the sole dependency on specifications, Li et al. [77] analyzed input fields and their surrounding texts in HTML pages to infer valid input conditions.

Code-based IVT methods [47, 79, 80] apply static analysis to extract valid/invalid input conditions from source code. Halfond et al. [47] also used symbolic execution for precise analysis of input conditions.

In general, although IVT approaches could detect some vulnerabilities, it is hard to be effective. Their ability to detect depends largely on the completeness of specifications and the adequacy of generated test suite for exposing input validation schemes that are weak against SQLI and XSS attacks.

Black-box vulnerability testing: There are many off-the-shelf black-box vulnerability scanning tools in the wild. Secubat [63] is an open source vulnerability scanning tool. It uses a web spider to identify test targets (e.g., web pages which accept user inputs). It then launches pre-defined attacks against those targets and concludes if an attack was successful by evaluating the server response against attack-specific response criteria (e.g., SQL exceptions raised and web page crashes). There are many other open source scanners like Secubat, such as Nikto2 (cirt.net/nikto2) and SQLMap (sqlmap.sourceforge.net/), which work in a similar fashion as Secubat; but they generally require known vulnerability patterns or user intervention to conclude successful attacks.

There are also black-box scanning tools that are only commercially available. Fonseca et al. [37] and Bau et al. [11] conducted a study on the performance of a few popular commercial vulnerability scanners such as HP WebInspect (fortify.com/products/web_inspect.html), IBM Rational AppScan (ibm.com/software/awdtools/appscan), and Acunetix Web Vulnerability Scanner (acunetix.com/vulnerability-scanner). In Bau et al.'s experiments, the tools detected only reflected-type XSSVs at a fairly good detection rate of 60%. The detection rate was only 0%-21.4% for other vulnerability types like stored XSS and second order SQLI.

Hence, being black-box based, these tools are generally not effective at detecting vulnerabilities. However, they could be extremely useful for quick detection of the presence of SQLI and XSS issues in websites. In contrast to these black-box testing approaches, our techniques presented in this thesis are all code-based (white-box) and are significantly more effective as the techniques check and analyze the vulnerability of every security-sensitive program statements in source code. But as pointed out by Goodenough and Gerhart [43] and Zhu et al. [169], black-box and white-box techniques complement each other since they test the systems on different aspects and therefore, a combination of them would provide more coverage.

White-box vulnerability testing: Shin et al. [139] used static analysis to track user inputs to database access points and generate unit test cases for these points. Test cases are crafted with SQLI attack patterns. There are also fault-based vulnerability testing techniques [38, 126, 127] in which vulnerabilities are injected into program source code and test cases containing adequate security attack vectors are generated and executed. Users are then to check if the injected vulnerabilities have been exposed by the test cases executed.

In general, the above approaches aim to generate adequate test suite for testing SQLIVs and/or XSSVs. However, these approaches are not designed to automatically detect vulnerabilities, that is, user intervention is required to inspect the test execution results and conclude the vulnerability. And the task of generating and injecting vulnerabilities is also manual for some of these techniques [126, 127].

7.5 Runtime Attack Prevention

These methods focus on preventing real-time attacks by using intrusion detection systems or runtime monitors. In general, these methods set up a proxy between application server and database server to intercept the queries sent from application server, or a proxy between client server and application server to intercept the incoming and outgoing HTTP traffic, and check for the illegal contents.

SQLI prevention methods: Major approaches that prevent SQLI attacks at runtime mainly involve two steps—(1) learning legitimate query structures; and (2) checking whether the query structures at runtime conform to the learned legitimate structures before those queries are sent to the database. Legitimate query structures are usually learned based on (1) user specification; (2) static analysis; or (3) dynamic analysis.

Kemalis and Tzouramanis [65] proposed a specification-based approach which requires developers to specify legitimate query structures using formal language expressions. Huang et al.'s work [56] is also dependent on user specification, and it also applies static analysis. They use type-based static analysis to identify potentially vulnerable code sections and instrument them with runtime guards. Users are required to specify pre-conditions of sensitive functions (e.g., functions which generate HTML outputs or database accesses) and post-conditions of sanitization functions applied. During runtime, instrumented guards check for the conformance of user-specified conditions. Pietraszek and Berghe [112] and Xu et al. [166] proposed dynamic analysis-based taint tracking approaches using metadata marking to track the flow of user inputs at runtime. They ensure that these inputs are syntactically confined (i.e., the input data is only treated as a literal value) or these inputs do not contain unsafe content defined in user-specified security policies. The approaches of Huang et al. [56], Pietraszek and Berghe [112], and Xu et al. [166] are applicable for both SQLI and XSS vulnerabilities.

Halfond and Orso [48] used static analysis to deduce queries that may legally appear at each database access point in web programs via isolation of tainted data and un-tainted data. Instead of targeting query statements in a server program like [48], Wei et al.'s work [161] targets stored procedures in database. They use a similar query learning approach as Halfond and Orso [48].

However, as pointed out by Bisht et al. [14], statically inferred legitimate query structures may not be accurate and attackers could exploit this weakness to conduct SQLIAs. As such, dynamic analysis-based approaches are later proposed to provide more accuracy. Buehrer et al. [18] and Su and Wassermann [144] track tainted data at runtime through marking them with meta-characters. When a query is invoked, its legitimate structure is learned by excluding marked data from the query. Conversely, Halfond et al.'s WASP [49] tracks un-tainted data motivated by the fact that it is often hard to identify all input sources in practice and some tainted data may be missed. These approaches require low user effort, but as meta-character marking changes the structure of original data, it may cause unpredictable errors on benign inputs.

Liu et al.'s SQLProb [78] executes the program with various valid inputs to collect all possible queries that may legally appear during runtime. It then uses a global pair-wise alignment algorithm to compare the issued user queries against those from collected legitimate query repository and extract user inputs. An extracted input is then validated (check whether it is part of the syntactic structure of the issued query from which it is extracted) using a SQL parser. Only if user input is syntactically confined, query is sent to the database. This approach requires users to exercise test inputs and assumes that test inputs used are sufficient to exercise all possible queries in the

program. Bisht et al.'s CANDID [14] dynamically mines legitimate query structure at each program path by executing the program with valid and non-attacking inputs and thereafter, it compares the actual issued query with the legitimate query structure mined for the same program execution path.

In contrast to the above SQLI prevention approaches, Boyd and Keromytis proposed a randomization technique [16] which enforces developers to construct queries using randomized SQL keywords instead of normal keywords. A proxy filter intercepts queries sent to the database and de-randomizes the keywords. An attacker could never inject SQL code without the secret key to randomization.

XSS prevention methods: Bisht and Venkatakrishnan's runtime monitor called XSS-Guard [13] transforms the server programs such that they produce a shadow web page for each real response page. Shadow page reflects the intended output of the program. Before sending the real response page to clients, XSS-Guard checks for the difference in script contents of real and shadow pages. Johns et al.'s XSSDS [60] also works in a similar way to XSS-Guard.

Some XSS prevention techniques such as Jim et al. [59] and Louw and Venkatakrishnan [83] require the collaboration of browsers. In Beep [59], browser is modified such that it is capable of detecting illegitimate scripts based on security policies sent from server programs and preventing those scripts from being executed. Unlike Beep, Blueprint [83] works on existing (un-modified) browsers. To evade the browsers' unreliable parsing behaviors, the Blueprint-installed-server takes over the browser's task of parsing tainted HTML contents (the part that might contain XSS vulnerability) and embeds the generated parse tree as a model in place of the actual tainted content. It also includes a model interpreter so that the browser is able to interpret the embedded models and produce the HTML documents as intended.

To provide a personal protection layer for clients and allow clients to avoid relying on the security of web application, some researchers [70, 145] proposed client-side solutions that defend against XSS. They install proxies on client-side browsers. Noxes [70] acts as a personal firewall which allows or blocks connections to websites based on filter rules. Filter rules are basically the whitelist and blacklist of URLs specified by users. Whenever the browser sends a HTTP request to an unknown website not listed in filter rules, Noxes immediately shows a connection alert to client who can then choose to permit or deny the connection and it remembers the client's action for future use. Sun et al. [145] detects XSS worms by checking the presence of any self-replicated scripts in the outgoing data sent by browsers.

The major advantage of runtime prevention techniques compared to our proposed techniques is that no approximation (of possible malicious data) is necessary because actual runtime values of inputs are checked. Therefore, in principal, these techniques can prevent most attacks. But, the drawback or the main difference from our techniques is that it requires installation of additional (possibly complex) frameworks to enable dynamic monitoring and it incurs runtime overhead due to interception of HTTP traffic. Furthermore, some of these approaches require users to define security policies (especially specification-based approaches), which usually cost a considerable amount of labor.

7.6 Summary

In this chapter, we have reviewed techniques for defending against SQLI and XSS exploits. These techniques can be classified into defensive coding practices, vulnerability prediction techniques, vulnerability detection techniques, vulnerability testing techniques, and runtime attack prevention techniques. Our review finds that these existing techniques suffer from one or more of the following weaknesses: (1) inherent limitations; (2) further developments required (3) incomplete implementations; (4) complex frameworks; (5) runtime overheads; (6) intensive manual work requirements.

Defensive coding practices offer non-vulnerable code, but they are labor-intensive and prone to human errors. Vulnerability prediction approaches help auditors focus on code locations that are likely to be vulnerable, but current prediction approaches locate vulnerable code at module or component level rather than program statement level. Vulnerability detection approaches could identify all vulnerabilities in principal, but they may generate many false alarms. Vulnerability testing approaches could generate adequate test suite for testing SQLI and XSS vulnerabilities in programs, but most of these testing approaches require much manual effort as users have to execute test cases and inspect execution results. There are also testing approaches that automate test suite generation and vulnerability testing, but they require automated execution frameworks whose implementations are often incomplete or unavailable to date. Attack prevention approaches could prevent all attacks during runtime but they require dynamic monitoring systems and incur runtime overheads.

Our techniques proposed in this thesis aim to address the shortcomings of these existing techniques. And we believe that an overlapped use of existing techniques and ours would certainly provide a complete defense mechanism against SQLI and XSS. Our vulnerability prediction techniques presented in Chapter 3 and Chapter 4 are easy to be used and could predict

vulnerabilities with high precision rates. Our vulnerability auditing technique presented in Chapter 5 could aid auditors in finding the flaws in the implementations of defensive code methods. Finally, our vulnerability removal method presented in Chapter 6 provides an automated way to fix the vulnerabilities. Furthermore, we also implemented prototype tools that automate our proposed approaches. The tools are publicly available in our website [116].

Chapter 8

CONCLUSIONS AND RECOMMENDATIONS

This chapter is organized as follows. Section 8.1 summarizes the contributions made by this thesis towards academic research. Section 8.2 discusses the contributions made by this thesis towards industry practice. Section 8.3 outlines some of the future research directions.

8.1 Contributions towards Academic Research

SQLI and XSS have been the two most common and serious security threats in recent years [29, 108]. Academic researchers have proposed a wide variety of solutions from simple static analysis techniques to complex runtime protection mechanisms to mitigate these two issues. But, until now, SQLI and XSS flaws continue to exist in many real-world web applications. We identified that current academic research works suffer from one or more of the following weaknesses, preventing them from comprehensively addressing these two security issues:

- 1) *Intensive manual work requirement*—Defensive coding practices are effective but, developers have to apply them manually. Static analysis-based approaches typically require user interventions to identify good and bad sanitization functions. Some approaches, especially specification-based, require users to manually define security policies or generate adequate test cases. Some approaches require users to manually inspect the test results to conclude the vulnerabilities. On the other hand, developers who adopt these approaches are often not well-trained in security. Thus, these existing research works are both labor-intensive and prone to errors.
- 2) *Coarse-grained analysis*—Existing vulnerability prediction approaches only predict vulnerabilities at software module, component, or program level. Hence, security auditors still have a lot of work to do to locate the vulnerable code sections.
- 3) *Inaccuracy*—Static analysis-based approaches are easy to implement and use. They are typically scalable. But, these approaches are often inaccurate in vulnerability detection

because static analysis can only provide general properties about programs or functions.

- 4) *In-scalability*—Unlike static approaches, dynamic analysis-based approaches are generally accurate in analyzing the properties of the functions. But, dynamic analysis frameworks such as model checkers and concolic execution engine are computationally expensive and not scalable. This is one of the main reasons why dynamic analysis approaches are rarely adopted in practice.
- 5) *Runtime overhead*—Runtime prevention approaches are the most efficient solution for already-deployed applications. But these approaches incur overheads in analyzing the scripts during runtime.
- 6) *Lack of program verification assistance*—Most of the existing approaches lack focus on reporting the defense features implemented in the programs. When vulnerability is detected, auditor or developer is required to fix it, which would be difficult without the information of implemented defense features.
- 7) *Incomplete implementation*—Some approaches might be comprehensive and very promising in addressing SQLI and XSS. But, they might fail in providing adequate implementations either commercially or publicly. And from our experiences, it requires tremendous effort and knowledge to implement the proposed theoretical concepts into practical vulnerability mitigation tools. It is difficult for developers who have little knowledge in security research to implement and adopt these approaches.

Hence, based on these observations, in this thesis, we presented three novel approaches—vulnerability prediction, vulnerability auditing, vulnerability removal, to defend against SQLI and XSS threats. Our research work mitigates the above-mentioned shortcomings of existing research works. The work presented in this thesis has been published in [128]-[136]. Summarizing our research work, its contributions towards academic research are:

- 1) *Minimal or no manual work requirement*—First, our vulnerability prediction approach like every other prediction approach requires users to gather past vulnerability data for training the predictors. But, once the training is done, predicting future vulnerabilities becomes fully automated, without any further user intervention required. Second, our vulnerability auditing approach is fully automated in extracting SQLI and XSS defense features. Although the auditing process is manual, it needs to be done only on the extracted defense features. In our auditing experiments, the extracted code is less than

10% of the total code (see Chapter 5.4.3). Thus, the manual work required is still minimal. Third, our vulnerability removal approach is fully automated in removing the vulnerabilities.

- 2) *Fine-grained analysis*—Our overall research focuses on analyzing the vulnerabilities at program statement level. Especially, in contrast to existing vulnerability prediction research, our vulnerability prediction approach represents each security-sensitive program statement in terms of a set of static and dynamic analysis-based software attributes that could indicate the vulnerability of that statement. As such, we can detect vulnerabilities at statement level. It is noteworthy that this improvement required us to research on a novel set of attributes completely different from those used in existing approaches, which typically rely on commonly-used software attributes such as code complexity to predict vulnerabilities.
- 3) *Accuracy*—To mitigate the accuracy issues associated with static-based approaches in detecting vulnerabilities, we proposed two novel solutions. First, we augmented static analysis with empirical knowledge and data mining techniques for predicting vulnerabilities, resulting in good accuracies (74% recall and 8% false alarm rate on average). Second, we enhanced our first solution with dynamic analysis, further improving the accuracies by 16% recall and 3% false alarm rate on average (see Chapter 4).
- 4) *Scalability*—We achieved scalability through the efficient use of empirical knowledge, program analyses, and data mining techniques. In detecting vulnerabilities, dynamic analysis techniques such as model checking and concolic execution are typically in-scalable because of state space explosion problem. Our vulnerability auditing and vulnerability removal approaches do not involve dynamic analysis. In our vulnerability prediction approach presented in Chapter 4, we use dynamic analysis only to infer the possible types of input validation and sanitization routines rather than to precisely prove their correctness through exploring numerous input spaces, and we apply data mining on those inferences for vulnerability prediction. Therefore, we mitigated the scalability issue typically associated with dynamic analysis.
- 5) *No runtime overhead*—Our research focuses on detecting and removing vulnerabilities from program source code. Unlike runtime prevention approaches, our work does not involve proxies and intrusion detection systems. Thus, there is no runtime overhead.

- 6) *Provide program verification assistance*—To assist security auditors in fixing the detected vulnerabilities, our research provides the techniques to systematically extract SQLI and XSS defense features implemented in code. Auditing guidelines are also provided. Using our work, auditors could identify the weakness of existing defense features implemented and the requirement of features not implemented.
- 7) *Provide complete implementation*—We developed the prototype tools, namely *PhpMiner*, *WAVDE*, and *saferXSS*, that implement our research proposals. The tools are provided in the author’s website [116].
- 8) *Experimental evaluation*—Using our prototype tools, we evaluated our research work comprehensively. We were able to predict almost all vulnerabilities from six real-world open source applications with an average accuracy of 90% recall and 5% false alarm rate (Chapter 4). In Chapter 5, we also performed XSS vulnerability audits on seven open source applications and SQLI vulnerability audits on five open source applications. We were able to verify and confirm all the vulnerabilities with zero false positives although the audits had to be manually performed on 10% of the total code. Last, as shown in Chapter 6, the *saferXSS* tool was able to automatically remove all the potential vulnerabilities from five open source applications.

Finally, our overall research can be concluded as follows. Considering that our auditing approach requires manual work, vulnerability auditing should be followed after the vulnerability prediction phase in order to detect the remaining vulnerabilities and filter the false warnings. Then, considering that our vulnerability removal approach relies on the potentially vulnerable code locations reported by an underlying vulnerability detection approach, conceptually, it should be applied after vulnerability prediction and auditing approaches in order to comprehensively address the threats of SQLI and XSS.

8.2 Contributions towards Industry Practice

In any industry, protecting the organization’s brand is an extremely important practice for its stakeholders. Commercial confidence and customer trust is critical to a business enterprise. In addition, protecting sensitive information is also an essential practice. Sensitive information includes personal, health, financial, or any other information that can affect the competitive edge of the organization. In the context of software industry, software security breaches may lead to critical situations such as disclosure of customer information and organization’s financial plans, denial of service, and threats to the continuity of business operations. While there may be adverse

financial consequences, the real cost to the organization will be the loss of customer trust and confidence in the organization's brand, and the loss of the organization's competitive edge.

Hence, software security is essential to the above two industry practices. This thesis contributes to these important practices since our work focuses on SQLI and XSS which are the two most common security threats to web applications nowadays.

To ensure secure software, a common industry practice is to think security since the early stages of the software development lifecycle. According to McGraw [89] and SAFECode [122], the following security best practices are well established in software industry, adopted by giant organizations such as Microsoft, SAP, and Symantec.

- 1) *Security Education*: Every stakeholder is responsible for IT security as a security breach could have dire consequences for everyone. Thus, software security should start from education. It is important to educate software developers, architects, and users about how to build secure software and how to use software securely.
- 2) *Secure Design*: Many security vulnerabilities are not due to coding faults but rather due to design issues such as business logic flaws. Thus, it is required to design software with secure features.
- 3) *Secure Code*: Developers need to follow the secure software design and incorporate necessary secure features into the code. It is also important to perform security code reviews from time to time.
- 4) *Security testing*: To ensure that secure design and secure coding guidelines were followed, security testing, such as vulnerability analysis and penetration testing, must be carried out.
- 5) *Security Documentation*: This is important to help customers understand how to use security controls and how to configure software in a secure way.
- 6) *Security Response*: Any security vulnerabilities (exploited or discovered) reported against the deployed software must be properly handled. Security patching and vulnerability mitigation attempts should be quickly carried out.
- 7) *Security Research*: Security auditors and developers need to be updated with latest security technologies and also new threat models so that they are aware of outdated security implementation methods and latest techniques to defend against new security attack vectors.

This thesis contributes to many of the above established industry practices for secure software in the following ways. First, since it provides the details of common SQLI and XSS vulnerabilities and also provides information on available state-of-the-art defense approaches, it could well educate security auditors and developers about the two common security threats and the effective defense methods. With this knowledge, developers would be able to design and develop software with secure features. Second, this thesis could also update security researchers in the industry with the latest technologies regarding SQLI and XSS. Third, through applying vulnerability auditing and vulnerability removal approaches proposed in this thesis, security auditors would be able to effectively review the code for vulnerabilities and fix them. Using our proposed vulnerability prediction techniques, a security tester could focus on testing the code predicted to be vulnerable, instead of spending much effort in testing the whole chunk of code.

Hence, to summarize, the information provided and the vulnerability mitigation approaches presented in this thesis could help software development professionals in ensuring the delivery of defect-free software, which is an industry practice that many of them aim to uphold dearly.

8.3 Recommendations

In this section, we identify key research areas and future work that could validate our current results or improve them and that could enhance our work presented in this thesis.

In our study, various experiments have been conducted to evaluate the proposed approaches. In the experiments, a number of test subjects have been used. Although our test subjects generally differ in functionalities and usages, and the sizes range from relatively small to large scales, these subjects used may still not be representative of all types of web applications.

Furthermore, the test subjects used are either Java-based or PHP-based systems. The main reason is due to the widespread of XSS and SQLI vulnerabilities in these systems, and also due to the readily available Java and PHP program analysis tools such as *Soot* and *Pixy*. However, the approaches presented in this thesis are either language-independent or easily extendible to fit the syntax of other programming languages.

Among the three types of XSS (reflected, stored, and DOM-based), our tools do not address DOM-based XSS because, like most other research tools, our tools can only analyze server programs. Addressing DOM-based XSS requires analysis of client-side scripts such as JavaScript and Flash. But this is mainly the limitation of the tool not the proposed approaches. If client script

analysis is possible in future, the same approaches proposed in this thesis can be used to address DOM-based XSS.

The above discussion exposes general limitations of our current work. Hence, clearly, an important future work is to address these limitations. Our experiments should be replicated and repeated using different set of systems (possibly industrial systems) to re-evaluate current findings. Especially in the context of vulnerability prediction study, many research works have observed that predictors built with code attributes do not achieve consistent results across systems. Researchers could do so as we have posted the experimented data and the prototype tools on our website [116].

Necessary modifications and enhancements of our proposed approaches and prototype tools need to be done to adapt to different programming languages such as .NET and client-side scripts. Then, these adapted approaches and modified tools should be re-evaluated to determine if our approaches can be generalized across programming languages. We recommend the following tasks to adapt our proposed three approaches to a different web programming language.

- 1) Since we provide algorithms of our approaches and/or architectures of our prototype tools in the thesis, one should use this information as the starting point for extending our work to a different language.
- 2) Many of the activities performed in our approaches can be exactly followed without any modification. For example, regarding vulnerability prediction approach, the same classification schemes proposed in this thesis could be applied to any web programming language. Also, there is no change required with respect to the proposed data mining activities to build predictors because the data mining tool, *Weka*, used in this thesis is language-independent. Similarly, in our vulnerability auditing approach, one could exactly follow the definitions which define various SQLI and XSS defense methods to extract their implementations from code and perform security audits using the proposed auditing rules. Regarding vulnerability removal approach, one could adopt the same XSS prevention rules and also the same security library, *ESAPI* (which supports multiple programming languages such as C, .NET, Python, etc.), used by us.
- 3) But, one major modification required is to use different program analysis tools corresponding to the languages to be adapted. This is not too difficult as our work is largely based on conventional control flow and data flow analysis and pattern matching. There are readily available program analysis tools for different languages. For example, for C/C++,

one could use *CodeSonar* by GrammaTech [45]. Microsoft also provides *FxCop* [94] for static analysis of .NET programs.

- 4) Another modification required is that one has to redefine the code patterns used in our approaches according to the context of the language to be adapted. For example, in Java, we identify the code `response.getWriter().out` as an HTML output statement whereas, in PHP, `echo` is an HTML output statement.

Another future research recommendation is to explore the applications of our proposed approaches in mitigating other types of web application vulnerabilities which have similar characteristics to SQLI and XSS. Vulnerabilities such as buffer overflow, path traversal, cross site request forgery, and URL redirect are some possibilities because, like SQLI and XSS, these vulnerabilities are also application-level vulnerabilities caused by inadequate handling of user inputs.

The following presents specific ideas for future work on each of our proposed approaches presented.

Our vulnerability prediction work could be further explored by the use of more data mining activities so as to unearth better vulnerability predictors. Moreover, as we discussed above, our proposed prediction methods could be extended to address other types of vulnerabilities. For example, program statements that perform string length check and array assignment might be important indicators of buffer overflow vulnerability. Thus, we can use code attributes that reflect such program statements to build buffer overflow vulnerability predictors.

In our vulnerability audition study, we suggested guidelines to manually audit the defense features extracted by the proposed approach. A necessary future work to be carried out is to evaluate the feasibility and usefulness of the proposed approach based on intensive experiments on both open-source and industrial web applications of larger sizes. This study is necessary because the scalability of the proposed approach would be in question due to the requirement of manual verification process to identify the vulnerabilities. However, we have stressed that this manual work is unavoidable when fixing the vulnerabilities. Researchers could find the possibilities of combining our work with other automated approaches to further reduce the workload on manual verification process. For example, one could automate this auditing process by implanting the proposed audition rules into the tool and using intelligent algorithms to check for the conformance of these rules in the code extracted.

In our vulnerability removal study, we secured the potentially vulnerable HTML output statements using appropriate escaping mechanisms provided by a security library called *ESAPI* [110]. As *EASPI* also provides many other security mechanisms, researchers could explore the applicability of our removal approach in mitigating other vulnerabilities by making full use of *ESAPI*'s capabilities.

Finally, while we believe that the proposed three approaches can be alternative and complementary solutions to existing vulnerability mitigating approaches, research works that investigate the feasibility and usefulness of integrating multiple approaches (e.g., defensive coding+prediction+detection+removal) need to be carried out to validate our claims. And this could prove to be an interesting research topic.

APPENDIX A

Table A-1. Notations used

Symbol	Meaning
SQL	structured query language
SQLI	SQL injection
XSS	cross site scripting
SQLIV	SQL injection vulnerability
XSSV	cross site scripting vulnerability
CFG	control flow graph
ICFG	interprocedural control flow graph
DDG	data dependence graph
CDG	control dependence graph
S	backward static program slice
TIFG	taint-based information flow graph
DBMS	database management system
LOC	lines of code
SQL statement	statement in the program that creates, reads, or modifies records in the database
HTML output statement	statement in the program that generates HTTP response output data
sink	security-sensitive program statement or operation
pv-out	potentially vulnerable output
IVT	input validation testing
pd	probability of detection or recall
pf	probability of false alarm
pr	precision
acc	accuracy
bal	balance
tp	true positive
tn	true negative
fp	false positive
fn	false negative
NB	Naïve Bayes classifier
C4.5	C4.5 classifier
MLP	Multi-Layer Perceptron classifier
LR	Logistic Regression classifier
PCA	principal component analysis

AUTHOR'S PUBLICATIONS

- [1] L.K. Shar and H.B.K. Tan, "Auditing the defense against cross site scripting in web applications," in *Proceedings of the 2010 International Conference on Security and Cryptography*, pp. 505-511, July 2010.
- [2] L.K. Shar and H.B.K. Tan, "Semi-automated verification and removal of open redirect vulnerabilities in web applications," in *Proceedings of the 2010 Software Engineering Research and Practice*, pp. 522-528, July 2010.
- [3] L.K. Shar and H.B.K. Tan, "Defending against cross-site scripting attacks," *IEEE Computer*, vol. 45, no. 3, pp. 55-62, Mar. 2012.
- [4] L.K. Shar and H.B.K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Information and Software Technology*, vol. 54, no. 5, pp. 467-478, May 2012.
- [5] L.K. Shar and H.B.K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1293-1296, June 2012 (**top conference**).
- [6] L.K. Shar and H.B.K. Tan, "Auditing the XSS defence features implemented in web application programs," *IET Software*, vol. 6, no. 4, pp. 377-390, Aug. 2012.
- [7] 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*, pp. 310-313, Sept. 2012.
- [8] L. Kaiping, H.B.K. Tan, and L.K. Shar, "Semi-automated verification of defense against SQL injection in web applications," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, Dec. 2012.
- [9] L.K. Shar and H.B.K. Tan, "Defeating SQL injection," *IEEE Computer*, vol. 46, no. 3, pp. 69-77, Mar. 2013.
- [10] 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 35th International Conference on Software Engineering*, pp. 642-651, May 2013 (**top conference**).
- [11] L.K. Shar and H.B.K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization code patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767-1780, Oct. 2013.

BIBLIOGRAPHY

- [1] </xssed>, “Vodafone.com XSS helps you trace unregistered ‘Pay As You Go’ subscribers,” <http://www.xssed.com/newslist>, May 2010.
- [2] F.E. Allen, “Control flow analysis,” *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1-19, July 1970.
- [3] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, no. 3, pp. 137-146, Mar. 1976.
- [4] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, Massachusetts, 2004.
- [5] N. Alshahwan and M. Harman, “Automated web application testing using search based software engineering,” *In Proceedings of the 26th International Conference on Automated Software Engineering*, pp. 3-12, Nov. 2011.
- [6] C. Anley, “Advanced SQL injection in SQL server applications,” *Next Generation Security Software Ltd.*, White Paper, 2002.
- [7] E. Arisholm, L.C. Briand, and E.B. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” *Journal of Systems and Software*, vol. 83, no. 1, pp. 2-17, Jan. 2010.
- [8] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, “Finding bugs in dynamic web applications,” *in Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 261-272, July 2008.
- [9] D. Balzarotti, M. Cova, V. Felmetzger, 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 IEEE Symposium on Security and Privacy*, pp. 387-401, May 2008.
- [10] V.R. Basili, L.C. Briand, and W.L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, Oct. 1996.
- [11] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, “State of the art: automated black-box web application vulnerability testing,” *in Proceedings of the IEEE Symposium on Security and Privacy*, pp. 332-345, May 2010.
- [12] K.J. Biba, “Integrity considerations for secure computer systems,” *Technical report ESD-TR-76-372*, USAF Electronic Systems Division, Bedford, MA, Apr. 1977.

- [13] P. Bisht and V.N. Venkatakrishnan, "XSS-Guard: Precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23-43, July 2008.
- [14] P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks," *ACM Transactions on Information and System Security*, vol. 13, no. 2, pp. 1-39, Feb. 2010.
- [15] M.D. Bond and K.S. McKinley, "Practical path profiling for dynamic optimizers" in *Proceedings of International Symposium on Code Generation and Optimization*, pp. 205-216, Mar. 2005.
- [16] S.W. Boyd and A.D. Keromytis, "SQLrand: preventing SQL injection attacks," *Lecture Notes in Computer Science*, vol. 3089, pp. 292-302, 2004.
- [17] L.C. Briand, J. Wüst, J.W. Daly, and D.V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245-273, May 2000.
- [18] 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*, pp. 106-113, Sept. 2005.
- [19] BugTraq, "BugTraq Mailing List Archive," <http://www.securityfocus.com/archive/1>, accessed Nov. 2010.
- [20] C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, pp.1066-1071, May 2011.
- [21] C. Catal, "Software fault prediction: a literature review and current trends," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626-4636, Apr. 2011.
- [22] S. Le Cessie, J.C. Van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191-201, 1992.
- [23] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: a survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1-58, July 2009.
- [24] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proceedings of the 18th international symposium on Software testing and analysis*, pp. 141-152, July 2009.
- [25] J. Clarke, *SQL injection attacks and defense*. 2nd ed., Syngress, 2012.

- [26] W.R. Cook and S. Rai, "Safe query objects: statically typed objects as remotely executable queries," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 97-106, May 2005.
- [27] CVE, "Common vulnerabilities and Exposures", <http://cve.mitre.org>, accessed Mar. 2012.
- [28] CWE, "CWE-79: Improper neutralization of input during web page generation," <http://cwe.mitre.org/data/definitions/79.html>, accessed June 2010.
- [29] CWE/SANS, "Top 25 most dangerous programming errors," <http://www.applicure.com/blog/cwe-sans-top-25-dangerous-programming-errors>, accessed Jun. 2010.
- [30] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1-30, Dec. 2006.
- [31] Eclipse, "An open development platform," <http://www.eclipse.org/>, 2010.
- [32] N.E. Fenton and S.L. Pfleeger, *Software metrics: a rigorous and practical approach*. 2nd ed., International Thompson Press, 1997.
- [33] N.E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.
- [34] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.
- [35] D. Fisher, L. Xu, and N. Zard, "Ordering effects in clustering," in *Proceedings of the 9th International Conference on Machine Learning*, pp. 163-168, 1992.
- [36] S. Fogie, J. Grossman, R. Hansen, and A. Rager, *XSS exploits: cross site scripting attacks and defense*. Syngress, pp. 395-406, 2007.
- [37] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pp. 365-372, Dec. 2007.
- [38] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & attack injection for web applications," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems & Networks*, pp. 93-102, June/July 2009.
- [39] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting SQL injection Vulnerability," in *Proceedings of the 31st International Conference on Computer Software and Applications Conference*, pp. 87-96, July 2007.

- [40] X. Fu and C.C. Li, "A string constraint solver for detecting web application vulnerability," in *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, pp. 535-542, July 2010.
- [41] K. Gao, T.M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software Practice and Experience*, vol. 41, no. 5, pp. 579-606, Apr. 2011.
- [42] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification through code-level metrics," in *Proceedings of the 4th ACM Workshop on Quality of Protection*, pp. 31-38, Oct. 2008.
- [43] J.B. Goodenough and S.L. Gerhart, "Toward a theory of test data selection," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 156-173, June 1975.
- [44] GotoCode, "Open source website," <http://www.gotocode.com>, accessed Sept. 2009.
- [45] GrammaTech, "CodeSonar: static analysis tool," <http://www.grammatech.com/codesonar>
- [46] Graphviz, "Graph visualization software," <http://www.graphviz.org/>, accessed Jan. 2010.
- [47] W. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pp. 285-296, July 2009.
- [48] W. 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*, pp. 174-183, Nov. 2005.
- [49] W. Halfond, A. Orso, and P. Manolios, "WASP: protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 65-81, Jan./Feb. 2008.
- [50] W. Halfond, J. Viegas, and A. Orso, "A classification of SQL injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Mar. 2006.
- [51] M. A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 6, pp. 1437-1447, Nov./Dec. 2003.
- [52] M. Halstead, *Elements of software science*. Elsevier, New York, 1977.
- [53] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. 3rd ed., Morgan Kaufmann, 2011.
- [54] J.H. Hayes and A.J. Offutt, "Input validation analysis and testing," *Empirical Software Engineering*, vol. 11, no. 4, pp. 493-522, Dec. 2006.

- [55] J.H. Hayes and A.J. Offutt, "Increased software reliability through input validation analysis and testing," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pp. 199-209, Nov. 1999.
- [56] 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*, pp. 40-52, May 2004.
- [57] E.J. Jackson, *A user's guide to principal components*. John Wiley & Sons Inc., 2003.
- [58] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: a concolic whitebox fuzzer for Java," in *Proceedings of the 1st NASA Formal Methods Symposium*, pp. 121-125, Apr. 2009.
- [59] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, pp. 601-610, May 2007.
- [60] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: server-side detection of cross-site scripting attacks," in *Proceedings of the Annual Computer Security Applications Conference*, pp. 335-344, Dec. 2008.
- [61] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 258-263, May 2006.
- [62] L. Kaiping, H.B.K. Tan, and L.K. Shar, "Semi-automated verification of defense against SQL injection in web applications," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, Dec. 2012.
- [63] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: a web vulnerability scanner," in *Proceedings of the 15th international conference on World Wide Web*, pp. 247-256, May 2006.
- [64] V. Katos, "Network intrusion detection: evaluating cluster, discriminant, and logit analysis," *Information Sciences*, vol. 177, no. 15, pp. 3060-3073, Aug. 2007.
- [65] K. Kemalis and T. Tzouramanis, "SQL-IDS: a specification-based approach for SQL-injection detection," in *Proceedings of the 23rd ACM Symposium on Applied Computing*, pp. 2153-2158, Mar. 2008.
- [66] T.M. Khoshgoftaar, N. Seliya, and N. Sundaresh, "An empirical study of predicting software faults with case-based reasoning," *Software Quality Journal*, vol. 14, no. 2, pp. 85-111, June 2006.

- [67] A. Kiežun, V. Ganesh, P.J. Guo, P. Hooimeijer, and M.D. Ernst, “HAMPI: a solver for string constraints,” in *Proceedings of the 18th International Symposium on Testing and Analysis*, pp. 105-116, July 2009.
- [68] A. Kiežun, 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*, pp. 199-209, May 2009.
- [69] J.C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, July 1976.
- [70] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, “Client-side cross-site scripting protection,” *Computers & Security*, vol. 28, no. 7, pp. 592-604, Oct. 2009.
- [71] A. Klein, “DOM based cross site scripting or XSS of the third kind,” <http://www.webappsec.org/projects/articles/071105.shtml>, July 2005.
- [72] B. Korel, “The program dependence graph in static program testing,” *Information Processing Letters*, vol. 24, no. 2, pp. 103-108, Jan. 1987.
- [73] M.S. Lam, M. Martin, B. Livshits, and J. Whaley, “Securing web applications with static and dynamic information flow tracking,” in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 3-12, Jan. 2008.
- [74] T. Lengauer and R.E. Tarjan, “A fast algorithm for finding dominators in a flowgraph,” *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121-141, July 1979.
- [75] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: a proposed framework and novel findings,” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485-496, July/Aug. 2008.
- [76] N. Li, J. Wu, M.-Z. Jin, and C. Liu, “Web application model recovery for user input validation testing,” in *Proceedings of the 2nd International Conference on Software Engineering Advances*, Aug. 2007.
- [77] N. Li, T. Xie, M. Jin, and C. Liu, “Perturbation-based user-input-validation testing of web applications,” *Journal of Systems and Software*, vol. 83, no. 11, pp. 2263-2274, Nov. 2010.
- [78] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, “SQLProb: a proxy-based architecture towards preventing SQL injection attacks,” in *Proceedings of the 24th ACM Symposium on Applied Computing*, pp. 2054-2061, Mar. 2009.
- [79] H. Liu and H.B.K. Tan, “Testing input validation in web applications through automated model recovery,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 222-233, Feb. 2008.

- [80] H. Liu and H.B.K. Tan, "Covering code behavior on input validation in functional testing," *Information and Software Technology*, vol. 51, no. 2, pp. 546-553, Feb. 2009.
- [81] B. Livshits, A.V. Nori, S.K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 75-86, June 2009.
- [82] B. Livshits and M.S. Lam, "Finding security errors in Java programs with static analysis," in *Proceedings of the 14th Usenix Security Symposium*, pp. 271-286, Aug. 2005.
- [83] M.T. Louw and V.N. Venkatakrishnan, "Blueprint: robust prevention of cross-site scripting attacks for existing browsers," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 331-346, May 2009.
- [84] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 416-426, May 2007.
- [85] M. Martin and M.S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *Proceedings of the 17th USENIX Security Symposium*, pp. 31-43, July 2008.
- [86] M. Martin, B. Livshits, and M.S. Lam, "Finding application errors and security flaws using PQL: a program query language," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 365-383, Oct. 2005.
- [87] T.J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.
- [88] R.A. McClure and I.H. Krüger, "SQL DOM: compile time checking of dynamic SQL statements," in *Proceedings of the 27th international conference on Software engineering*, pp. 88-96, May 2005.
- [89] G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80-83, Apr. 2004.
- [90] T. Mende, "Replication of defect prediction studies: problems, pitfalls and recommendations," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp. 1-10, Sept. 2010.
- [91] 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.
- [92] 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 Engineering*, vol. 17, no. 4, pp. 375-407, Dec. 2010.

- [93] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, pp. 47-54, May 2008.
- [94] Microsoft, "FxCop," [http://msdn.microsoft.com/en-US/library/bb429476\(v=vs.80\).aspx](http://msdn.microsoft.com/en-US/library/bb429476(v=vs.80).aspx)
- [95] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th International Conference on World Wide Web*, pp. 432-441, May 2005.
- [96] D.C. Montgomery and G.C. Runger, *Applied statistics and probability for engineers*. J. Wiley, New York, 2003.
- [97] A. Mueller, "Cross site scripting (XSS)," <http://elegantcode.com/2009/05/28/cross-site-scripting-xss/>, May 2009.
- [98] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 284-292, May 2005.
- [99] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 580-586, May 2005.
- [100] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pp. 62-74, Nov. 2006.
- [101] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 529-540, Oct. 2007.
- [102] M.N. Ngo and H.B.K. Tan, "Empirical-based recovery and maintenance of input error-correction features," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 6, pp. 419-450, Oct. 2007.
- [103] NVD, "National vulnerability database," <http://nvd.nist.gov/>, accessed Nov. 2012.
- [104] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 187-197, Nov. 2004.
- [105] A. Orso, S. Sinha, and M.J. Harrold, "Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 199-239, Apr. 2004.

- [106] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340-355, Apr. 2005.
- [107] OWASP, "SQL injection prevention cheat sheet," https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, accessed Feb. 2010.
- [108] OWASP, "The ten most critical web application security vulnerabilities," <http://www.owasp.org/>, accessed Feb. 2010.
- [109] OWASP, "XSS prevention cheat sheet," [http://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet), accessed Jan. 2010.
- [110] OWASP, "Enterprise security API," https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, accessed Jan. 2010.
- [111] S. Palmer, *Web application vulnerabilities: detect, exploit, prevent*. Elsevier, 2007.
- [112] T. Pietraszek and C.V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," *Lecture Notes in Computer Science*, vol. 3858, pp. 124-145, 2006.
- [113] L. Portnoy, E. Eskin, and S. Stolfo, "Intrusion detection with unlabeled data using clustering," in *Proceedings of the ACM CSS Workshop on Data Mining Applied to Security*, 2001.
- [114] PROMISE, "Software engineering repository," <http://promise.site.uottawa.ca/SERepository/>, accessed July 2011.
- [115] R. Quinlan, *C4.5: Programs for machine learning*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.
- [116] Research on web security testing, <http://sharlwinkhin.com/>
- [117] W. Robertson and G. Vigna, "Static enforcement of web application integrity through strong typing," in *Proceedings of the 18th USENIX Security Symposium*, pp. 283-298, Aug. 2009.
- [118] RSnake, "SQL injection cheat sheet Esp: for filter evasion," <http://ha.ckers.org/sqliinjection/>, accessed Mar. 2012.
- [119] RSnake, "XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion," <http://ha.ckers.org/xss.html>, accessed Mar. 2010.
- [120] M. Ruse, T. Sarkar, and S. Basu, "Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs," in *Proceedings of the 10th IEEE/IPSJ International Symposium on Applications and the Internet*, pp. 31-37, July 2010.

- [121] J.R. Ruthruff, A.S. Elbaum, and A.G. Rothermel, "Experimental program analysis: a new program analysis paradigm," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 49-60, July 2006.
- [122] SAFECODE, "Software assurance: an overview of current industry best practices," *white paper*, http://www.safecode.org/publications/SAFECODE_BestPractices0208.pdf, Feb. 2008.
- [123] N.F. Schneidewind, "Methodology for validating software metrics," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 410-422, May 1992.
- [124] N. Seliya and T.M. Khoshgoftaar, "Software quality analysis of unlabeled program modules with semisupervised clustering," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 37, no. 2, pp. 201-211, Mar. 2007.
- [125] K. Sen and G. Agha, "CUTE and jCUTE: concolic unit testing and explicit path model-checking tools," in *Proceedings of the 18th Conference on Computer Aided Verification*, pp. 419-423, Aug. 2006.
- [126] H. Shahriar and M. Zulkernine, "MUSIC: mutation-based SQL injection vulnerability checking," in *Proceedings of the 8th International Conference on Quality Software*, pp. 77-86, Aug. 2008.
- [127] H. Shahriar and M. Zulkernine, "MUTEC: mutation-based testing of cross site scripting," in *Proceedings of the 5th International Workshop on Software Engineering for Secure Systems*, pp. 47-53, May 2009.
- [128] L.K. Shar and H.B.K. Tan, "Auditing the defense against cross site scripting in web applications," in *Proceedings of the 2010 International Conference on Security and Cryptography*, pp. 505-511, July 2010.
- [129] L.K. Shar and H.B.K. Tan, "Semi-automated verification and removal of open redirect vulnerabilities in web applications," in *Proceedings of the 2010 Software Engineering Research and Practice*, pp. 522-528, July 2010.
- [130] L.K. Shar and H.B.K. Tan, "Defending against cross-site scripting attacks," *IEEE Computer*, vol. 45, no. 3, pp. 55-62, Mar. 2012.
- [131] L.K. Shar and H.B.K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Information and Software Technology*, vol. 54, no. 5, pp. 467-478, May 2012.
- [132] L.K. Shar and H.B.K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1293-1296, June 2012.

- [133] L.K. Shar and H.B.K. Tan, "Auditing the XSS defence features implemented in web application programs," *IET Software*, vol. 6, no. 4, pp. 377-390, Aug. 2012.
- [134] 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*, pp. 310-313, Sept. 2012. © ACM, 2012. The presented version is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in <http://doi.acm.org/10.1145/2351676.2351733>
- [135] L.K. Shar and H.B.K. Tan, "Defeating SQL injection," *IEEE Computer*, vol. 46, no. 3, pp. 69-77, Mar. 2013.
- [136] 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 35th International Conference on Software Engineering*, pp. 642-651, May 2013.
- [137] M. Shepperd and D.C. Ince, "A critique of three metrics," *Journal of Systems and Software*, vol. 26, no. 3, pp. 197-210, Sept. 1994.
- [138] Y. Shin, A. Meneely, L. Williams, and J.A. 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./Dec. 2011.
- [139] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: SQL injection testing using static and dynamic analysis," in *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering*, Nov. 2006.
- [140] S. Sinha, M.J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 209-254, Apr. 2001.
- [141] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356-370, May/June 2011.
- [142] Soot, "A Java optimization framework," <http://www.sable.mcgill.ca/soot/>, accessed Feb. 2009.
- [143] SourceForge, "Open source website," <http://www.sourceforge.net>, accessed Sept. 2009.
- [144] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proceedings of the 33rd Symposium on Principles of Programming Languages*, pp. 372-382, Jan. 2006.

- [145] F. Sun, L. Xu, and Z. Su, "Client-side detection of XSS worms by monitoring payload propagation," in *Proceedings of the 14th European Symposium on Research in Computer Security*, pp. 539-554, Sept. 2009.
- [146] S. Thamaraiselvi, R. Srivathsan, J. Imayavendhan, R. Muthuregunathan, and S. Siddharth, "Combining naive-bayesian classifier and genetic clustering for effective anomaly based intrusion detection," *Lecture Notes in Computer Science*, vol. 5908, pp. 455-462, Dec. 2009.
- [147] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities," *Information and Software Technology*, vol. 51, no. 3, pp. 589-598, Mar. 2009.
- [148] A. Tosun, A. Bener, and R. Kale, "Ai-based software defect predictors: applications and benefits in a case study," in *Proceedings of the 22nd Innovative Applications of Artificial Intelligence Conference*, pp. 1748-1755, July 2010.
- [149] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pp. 1-9, May 2009.
- [150] B. Turhan, G. Kocak, and A. Bener, "Data mining source code for locating software bugs: a case study in telecommunication industry," *Expert Systems with Applications*, vol. 36, no. 6, pp. 9986-9990, Aug. 2009.
- [151] US-CERT, "Microsoft .NET framework contains a cross-site scripting vulnerability," <http://www.kb.cert.org/vuls/id/455604>, Oct. 2010.
- [152] W. Visser, C.S. Păsăreanu, and S. Khurshid, "Test input generation with java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 97-107, July 2004.
- [153] L.G. Votta and A.A. Porter, "Experimental software engineering: a report on the state of the art," in *Proceedings of the 17th International Conference on Software Engineering*, pp. 277-279, 1995.
- [154] W3C, "HTML 4.01 specification," <http://www.w3.org/TR/html401/>, 1999.
- [155] W3C, "XHTML 1.0 specification," <http://www.w3.org/TR/xhtml1/>, 2002.
- [156] J. Walden, M. Doyle, G.A. Welch, and M. Whelan, "Security of open source web applications," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 545-553, Oct. 2009.
- [157] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, Sept. 2007.

- [158] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 171-180, May 2008.
- [159] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 32-41, June 2007.
- [160] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 249-260, July 2008.
- [161] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing SQL injection attacks in stored procedures," in *Proceedings of the Australian Software Engineering Conference*, pp. 191-198, Apr. 2006.
- [162] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, pp. 439-449, Mar. 1981.
- [163] I.H. Witten and E. Frank, *Data Mining*. 2nd ed., Morgan Kaufmann, 2005.
- [164] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 359-368, June/July 2009.
- [165] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th USENIX Security Symposium*, pp. 179-192, July/Aug. 2006.
- [166] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," in *Proceedings of the 15th USENIX Security Symposium*, pp. 121-136, July/Aug. 2006.
- [167] A. Yip, X. Wang, N. Zeldovich, and M.F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 291-304, Oct. 2009.
- [168] S. Zhong, T.M. Khoshgoftaar, and N. Seliya, "Unsupervised learning for expert-based software quality estimation," in *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering*, pp. 149-155, Mar. 2004.
- [169] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software unit test coverage and adequacy," *ACM computing surveys*, vol. 29, no. 4, pp. 366-427, Dec. 1997.
- [170] T. Zimmermann and N. Nagappan, "Predicting defect using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 531-540, May 2008.