A Recommender System for Improving Program Security Through Source Code Mining and Knowledge Extraction

Fitzroy Nembhard
A Recommender System for Improving Program Security Through Source Code Mining and Knowledge Extraction

by

Fitzroy Nembhard

Master of Science
Bioinformatics
Morgan State University
2012

Bachelor of Science
Computer Science
Morgan State University
2009

A dissertation submitted to Florida Institute of Technology in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Melbourne, Florida July, 2018
The author grants permission to make single copies.
We the undersigned committee
hereby approve the attached dissertation

A Recommender System for Improving Program Security Through Source Code Mining and Knowledge Extraction by Fitzroy Nembhard

Marco M. Carvalho, Ph.D.
Professor/Dean
College of Engineering and Computing
Committee Chair

Carlos Otero, Ph.D
Associate Professor
Electrical/Computer Engineering
Outside Committee Member

Thomas C. Eskridge, Ph.D.
Associate Professor
School of Computing
Committee Member

William H. Allen, III, Ph.D
Associate Professor
School of Computing
Committee Member

Philip J. Bernhard, Ph.D.
Associate Professor and Director
School of Computing
ABSTRACT

Title: A Recommender System for Improving Program Security Through Source Code Mining and Knowledge Extraction

Author: Fitzroy Nembhard

Major Advisor: Marco M. Carvalho, Ph.D.

The security of computer programs and systems is a very critical issue. Threats against computer networks and software are on the rise. Therefore, businesses and IT professionals should take steps to ensure that their information systems are as secure as possible. However, many programmers and software engineers do not think about adding security to their programs until their projects are near completion, which results in vulnerable and insecure systems that could be exploited by attackers.

This dissertation presents a recommender system to help programmers write more secure code. We created a model that mines and categories source code from existing open source projects and uses machine learning and text mining techniques to offer recommendations and example fixes to programmers of where security could be included in their projects. We achieved positive results in the performance and statistically significant results regarding the usability and the ability of the system to help programmers write more secure code.
# Table of Contents

Abstract iii

List of Figures x

List of Tables xii

Acknowledgments xiii

Dedication xv

1 Introduction 1

1.1 Important Definitions 2

1.2 The Problem 3

1.3 Research Question 5

1.4 The Solution: Recommending Security 6

1.5 Dissertation Structure 7

2 Literature Review 8

2.1 SAST Approaches 8

2.1.1 Lexical Analysis 9

2.1.2 Actionable Alert Identification Techniques (AAIT) 10

2.1.2.1 Alert Type Selection 10
2.1.2.2 Contextual Information .................................. 11
2.1.2.3 Data Fusion ............................................. 12
2.1.2.4 Graph Theory ........................................... 13
2.1.2.5 Machine Learning ........................................ 14
2.1.2.6 Dynamic Detection ...................................... 16
2.1.2.7 Model Checking .......................................... 18
2.1.3 Clustering .................................................. 19
2.2 DAST Approaches ............................................. 21
  2.2.1 Automated Software Test Case Generation ................... 22
    2.2.1.1 Symbolic Execution .................................. 22
    2.2.1.2 Model-based Testing .................................. 26
    2.2.1.3 Combinatorial Testing ................................ 28
    2.2.1.4 Adaptive Random Testing .............................. 29
    2.2.1.5 Search-Based Testing ................................. 30
2.3 Hybrid Analysis .............................................. 32
2.4 Related Work ................................................ 33
  2.4.1 Mining topic models from source code ...................... 33
  2.4.2 Machine Learning/AI Systems ................................ 34
  2.4.3 Code Completion ......................................... 34
  2.4.4 Difference Between our Approach and Existing
       Approaches ................................................ 35

3 Proposed Approach ............................................. 36
  3.1 Overview of Approach ...................................... 36
  3.2 The Data Analyzer ......................................... 37
  3.3 The Classification System .................................. 37
3.4 The Recommender System ........................................ 38

4 Data Understanding .................................................. 39
  4.1 The NVD/CVE ..................................................... 39
  4.2 The Sourcerer 2011 Dataset ...................................... 40
  4.3 Data Collection ................................................... 42

5 Data Analyzer Environment Setup ................................ 45
  5.1 Installing and Configuring Apache Hadoop for Running MapReduce Tasks ........................................... 45
    5.1.1 Step 1: Preliminary Checks and Hadoop Installation ....... 47
    5.1.2 Step 2: Host File Configuration and Key Generation ....... 47
    5.1.3 Step 3: Hadoop Configuration ................................ 48
    5.1.4 Step 4: System Verification ................................... 49

6 Modeling and Classification ........................................ 51
  6.1 Data Representation .............................................. 52
  6.2 Feature Extraction ................................................ 54
    6.2.1 MapReduce Algorithm For Feature Extraction ............ 57
    6.2.2 Extracting Features for Classifying CWE/CVE Vulnerabilities ............................................. 59
      6.2.2.1 CWE-89 – Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’) 60
      6.2.2.2 CWE-78 – Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’) 65
  6.3 Preparing Training Data .......................................... 68
6.4 Classifiers

6.4.1 Decision Trees

6.4.2 Random Forests

6.4.3 Support Vector Machines

7 System Design and Implementation

7.1 Initial System Design

7.2 Knowledge Elicitation Survey

7.2.1 Participants

7.2.2 Familiarity with Programming Languages and IDEs

7.2.3 Use of Existing Code Analyzers

7.2.4 Views and Expectations Regarding the Proposed tool that IntelliSenses Vulnerabilities

7.2.5 Themes that Emerged from the Survey

7.3 System Architecture

7.4 Final System Design

7.5 Recommending Fixes

7.5.1 Cosine Similarity

7.5.2 MinHash

7.5.3 SimHash

8 Evaluation

8.1 Classifier Performance

8.2 Scalability

8.3 Usability Study

8.3.1 Study Goal
D Usability Study Tasks 137

D.1 Overview ........................................... 137
  D.1.1 FindBugs ...................................... 138
  D.1.2 VulIntel ...................................... 138
D.2 General Demographic Questions .......................... 139
D.3 How to Use the Tools ................................ 139
  D.3.1 FindBugs ...................................... 139
  D.3.2 VulIntel ...................................... 139
D.4 TASK 1: SQL Injection ................................ 140
D.5 TASK 2: Command Injection ............................ 141

E Usability Study Questions 142

E.1 General Demographic Questions ......................... 142
E.2 Post-Task Completion Questionnaire .................... 143
E.3 Post-Task Completion Interview ........................ 145

F List of Publications 146
List of Figures

1.1 Summary of software failures in news articles in 2016 . . . . . . . . . 4
1.2 The average total cost of a data breach in 2017 compared to a four-
year average (*data not available for all years) . . . . . . . . . . . 5

2.1 The inclusion of static analysis tools within the software
development life cycle [1] . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Secure software development life cycle showing the inclusion of both
SAST and DAST. [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1 Overview of solution framework . . . . . . . . . . . . . . . . . . . . . 37

4.1 NVD XML 2.0 Schema . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.2 File Structure of the Sourcerer 2011 Repository . . . . . . . . . . 42
4.3 Number of vulnerabilities in the NVD 2017 List that were caused
by the top 10 SANS/CWE of 2011. The plot also shows the CWE
severity score for each CWE. . . . . . . . . . . . . . . . . . . . . . . . . 44

5.1 Snapshot of the VINE Web Interface . . . . . . . . . . . . . . . . . . 47

6.1 The model building phase . . . . . . . . . . . . . . . . . . . . . . . 51
6.2 ANTLR AST Example . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.3 JavaParser AST Example . . . . . . . . . . . . . . . . . . . . . . . . . 56
List of Tables


6.1 Features for recognizing SQL Injection . . . . . . . . . . . . . . . 63
6.2 Features for Recognizing OS Command Injection . . . . . . . . . . 68
6.3 Distribution of projects in part “A” of the Sourcerer dataset . . . 69
6.4 Breakdown of training and testing data . . . . . . . . . . . . . . . . 70

8.1 Classifier performance for SQLI . . . . . . . . . . . . . . . . . . . . 87
8.2 Classifier performance for command injection . . . . . . . . . . . . 87
8.3 Time taken to classify various open source projects for SQLI . . . 89
8.4 Two-way ANOVA regarding tool helpfulness . . . . . . . . . . . . . 98
8.5 Two-way ANOVA regarding tool usability . . . . . . . . . . . . . . . 98
8.6 Two-way ANOVA regarding the ability of the tools in helping users
write more secure code . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
8.7 Two-way ANOVA regarding tool adoption . . . . . . . . . . . . . . 100
Acknowledgements

I must express sincere gratitude to my advisor Dr. Marco Carvalho for his invaluable support all these years. He has gone above and beyond to help me not only academically but also adding that “human touch” that personifies the Florida Tech motto.

To the members of my committee, I say thank you for gladly accepting my request to serve in this capacity and for the tremendous input you have provided. Dr. Thomas Eskridge, thank you for being there as a strong source of support since I started working with the Harris Institute and even more so as I conducted research to complete this dissertation.

I must give special thanks to Dr. Heather Crawford for the invaluable guidance she provided with conducting the formative and usability studies presented in this work. Thanks go to the people who gladly participated in the survey and usability study. Thank you, Dr. Lucas Stephane and Dr. Nezamoddin Nezamoddini-Kachouie, for providing tips on statistical analysis in a usability context. I also express gratitude to Rosalyn Edwards for being such a kind and helpful student coordinator and friend.

Sincere gratitude is expressed to several members of the Bethel Abundant Life Ministries, especially Pastor Valvern Wittock, for providing support and for
connecting me with a person I have come to call a mother, Hilleret Knight-Watson. She has gone above and beyond to help me, and I will forever be indebted to her. Thanks also to Patricia Gardner, the Barnett family and the Swaby family, who provided spiritual encouragement and other support when the road got rough.

Words would fail me to mention by name everyone who has supported me throughout my studies at Florida Tech. But I must thank Cheryl Nicholson for providing encouragement even before I entered the university. Without all the support from everyone, I would not have been able to arrive at this milestone.
Dedication

I wish to dedicate this work to Hilleret Knight-Watson. You have played the role of a mother to an informally adopted son so well and I appreciate all you have done for me. Thanks, Mommy Watson.
Chapter 1

Introduction

Source code analysis involves analyzing program code for bugs or vulnerabilities and reporting this information to programmers, so they can take steps to mitigate coding defects. Tools that offer automated source code analysis have been around as early as the 1970s [4, 5, 6]. The majority of these tools provide static or dynamic analysis. Static analysis is “the process of evaluating a system or component based on its form, structure, content, or documentation” [7] while dynamic analysis involves performing analysis on a program while it is being executed [8]. It is a known fact that both static and dynamic analysis tools that find common bugs have high false positive and false negative rates that affect their adoption into the software engineering community [9, 10, 11, 12]. Therefore, for the past few decades, researchers have spent a great deal of effort proposing techniques for handling and improving the accuracy of the alerts reported by code analysis tools [13].

In the area of dynamic analysis, automated test case generation tools have been proposed to automatically select test cases that will meet certain testing objectives.

Hybrid approaches that combine the results of static analysis and dynamic analysis to catch errors missed by one type of analysis have been proposed [14, 15, 16]. However,
researchers argue that traditional hybrid approaches do not work well due to the
time required to run both analyses and combine results [17]. Researchers have also
investigated whether the experience of developers helps in detecting false positives
missed by code analyzers, and their conclusion is that there is “no improvement in the
classification with increased development experience or any individual experiences that
help the developers in detecting the false positives”[18]. This helps to underscore the
difficulty of finding actionable alerts to report to developers. It is, therefore, pertinent
that the source code analysis community endeavor to utilize new/unexplored techniques
to design better tools.

1.1 Important Definitions

Since this work focuses on designing and evaluating a system for improving program
security, it is necessary that we define preliminary terms that are used in this work. This
section is not exhaustive; other definitions are provided as they are used throughout the
text.

MITRE defines vulnerability as “a weakness in the computational logic found in
software and some hardware components that, when exploited, results in a negative
impact to confidentiality, integrity, or availability of a resource or system” [19].
Confidentiality, integrity and availability constitute what is known as the CIA-triad
(usually represented by a Venn diagram). A system is said to be secure if it satisfies all
three areas of the triad.

Additionally, an exposure is “a system configuration issue or a mistake in software
that allows access to information or capabilities that can be used by a hacker as a channel
into a system or network.” [19].

A fault is “an incorrect step, command, process, or data definition in a computer
program, design, or documentation.” [20].

A failure is “a departure from a system’s required behavior.” [20]
A bug is a general term that is defined based on context as 1) a mistake made by a human in interpreting a requirement, 2) a syntax error in program code, or 3) an unspecified error that results in a system crash [20].

### 1.2 The Problem

#### Problem Statement

There is a significant deficit of and need for tools that help programmers write secure code. Emphasis is often placed more on productivity than on security. Many programmers ignore security in order to deliver a program that solves a problem and meets a deadline. There is a plethora of static and dynamic analyzers available. However, developers are skeptical of using these tools because of the signal-to-noise ratio and time required to investigate inactionable alerts. In addition, many existing code analysis tools focus on finding bugs, but are not mitigative or predictive in helping programmers write secure code, and some systems have usability issues. In this research, we develop a recommender system that learns from open source projects, analyzes a user’s program code, and makes recommendations of secure practices, which, if followed, could result in more secure code.

We begin this work by providing a statement of the main problem we address in this dissertation. It has been a common truism for many decades that delayed issues in a program (known as the delayed issue effect or DIE) are harder to resolve than other errors [21]. However, Menzies et al. found no evidence of the delayed issue effect after analyzing 171 software projects that were developed between 2006 - 2014 [22]. The authors attribute the lack of DIE to the adoption of advanced technologies and practices that have enabled faster changes to software. These new advances include agile development, virtualization, cloud architectures, more CPU processing power, etc. Interestingly, the authors concluded that “the delayed issue effect may continue to be prevalent in some cases, such as high assurance software, architecturally complex systems, or in projects with poor engineering discipline”.

3
We do not argue that delayed issues are “harder” to resolve than other errors. Conversely, we argue that significant costs are imposed on individuals and organizations if vulnerabilities are not mediated as early as possible in program code. Research confirm that code-level vulnerabilities in application code cause increasingly many exploits [23, 24].

A 2017 report by Tricentis[25] shows that general software bugs account for a majority of software failures in 2016 while security vulnerabilities ranked second. The researchers analyzed 1,159 news articles and discovered 548 software fails (432 software bugs, 78 security vulnerabilities, and 38 usability glitches) in 363 companies. A monthly summary of these failures is provided in Figure 1.1. As can be seen in the figure, for 11/12 months in 2016, there were at least three security-related software failures reported in the news.

The 2017 Cost of Data Breach Study conducted by the Ponemon Institute shows that the average total cost of a data breach is US$3.62 million[26]. While the study showed that there has been and average of 10% decline in costs when compared with prior years, current costs are still staggering as shown in Figure 1.2.

Figure 1.1: Summary of software failures in news articles in 2016

The 2017 Cost of Data Breach Study conducted by the Ponemon Institute shows that the average total cost of a data breach is US$3.62 million[26]. While the study showed that there has been and average of 10% decline in costs when compared with prior years, current costs are still staggering as shown in Figure 1.2.
Figure 1.2: The average total cost of a data breach in 2017 compared to a four-year average (*data not available for all years)

The researchers who conducted the study recommend security analytics as one of the solutions that may decrease the cost of data breaches. Our premise is that the enforcement of good programming practices will result in less bugs or vulnerabilities making it into the release stages of software, which will ultimately reduce the costs (e.g. mitigation costs and lawsuits) that are imposed on organizations.

1.3 Research Question

The purpose of this research is to propose, design, and evaluate a recommender system that monitors active program code and offers suggestions to programmers of how to address unsafe practices as they code. Our goal is to show that useful knowledge can be
extracted from source code and used to build a model to aid programmers in creating more secure software. This results in the following research question:

---

**Research Question**

How applicable are machine learning and text mining in creating a recommender system that provides live security advice to programmers that will improve the security of the program being developed?

---

1.4 **The Solution: Recommending Security**

Recommender systems are software tools and techniques that provide suggestions for items that are most likely of interest to a particular user [27]. Recommender systems have been traditionally applied to commodities such as books, CDs, etc. Ricci et al. noted that the attributes of the items recommended by classic content-based recommendation techniques are keywords extracted from the descriptions of the items [27]. In this work, we propose to mine code repositories for code examples that mitigate certain vulnerabilities and use the classic recommendation approach to present code examples to the programmer that are most similar to the code being developed instead of using generic examples.

Source code mining is not a new concept. Researchers have used source code mining to detect, trace, and monitor architectural tactics in code [28]. In addition, researchers have also mined source code repositories in order to create a bug detector (function return value checker), which determines if a function’s return value is tested before being used [29]. Moreover, some researchers have also used machine learning techniques to identify actionable alerts from code analyzers to report to developers. (We discuss these approaches in detail in the literature review in Chapter 2). However, the application of source code mining to create recommender systems to help programmers write secure
code has not been explored. This observation is the basis for the ideas we propose in this work.

Our research answers questions such as:

1. How applicable is source code mining in building recommender systems?

2. To what extent can static analysis be augmented with IntelliSense technology based on knowledge extracted from open source projects?

3. How useful is a recommender system in helping programmers write more secure code?

1.5 Dissertation Structure

The dissertation continues with a literature review of program analysis approaches, including a synopsis of more closely related works, in Chapter 2. Chapter 3 presents an overview of the solution proposed in this work. In Chapter 4, we describe the datasets used in the work while Chapter 5 explains the process of setting up the experimental environment for data analysis and preprocessing. Chapter 6 provides a thorough discussion of the modeling phase. In Chapter 7, we discuss the design and implementation of the proposed system. This chapter also presents the results of a knowledge elicitation survey that was conducted to obtain information that affect the design and usability of the system. Finally, the work is evaluated in Chapter 8 followed by conclusions and recommended future work in Chapter 9.
Chapter 2

Literature Review

This chapter provides a literature review of different approaches that aim to improve the security of computer programs by means of code analysis. Section 2.1 provides a review of literature in the area of Static Application Security Testing (SAST). Section 2.2 discusses work in the area of Dynamic Application Security Testing (DAST) and Section 2.3 deals with hybrid analysis. We conclude the chapter with an overview of more current work that extracts topic models from source code, use AI to learn and reason about numeric constraints in programs, and work that applies code completion to help programmers become more productive.

2.1 SAST Approaches

In this section, we will review literature in the area of static analysis. Due to the challenge of finding the right people to do code reviews as well as its time consuming nature, researchers have included static code analysis tools within the software development life cycle (SDLC) to help mitigate coding defects. Figure 2.1 depicts the SDLC and shows the placement and importance of static analysis tools in security testing.
Figure 2.1: The inclusion of static analysis tools within the software development life cycle [1]

Static analysis approaches include lexical analysis, alert identification and alarm handling techniques. Alert identification techniques are based on prioritization and classification models, which include alert type selection, the use of contextual information to improve static alarms, data fusion, graph theory, machine learning, mathematical and statistical models, dynamic detection, and model checking. The same techniques used for prioritization are also used for classification [13]. Therefore, we will focus on classification techniques. In [30], Muske and Serebrenik discussed the use of ranking, clustering, and pruning techniques to handle static analysis alarms. However, many of these approaches can be collapsed under the classification techniques we discuss in this section (e.g. mathematical and statistical models, machine learning, and graph theory).

2.1.1 Lexical Analysis

The most basic approach to static analysis is known as lexical analysis. This is a rule-based approach, which involves preprocessing and tokenizing program source code and then attempting to match the resulting token stream against a database of faulty
program code [1]. Tools that fall into this category include ITS4\(^1\), FlawFinder\(^2\) and RATS\(^3\). While these tools can help locate specific bugs within a program, they are often considered lightweight and limited in scope and, therefore, not applicable to large, diverse projects. Consequently, a number of other approaches have been proposed. These approaches include developing sound and complete algorithms, using annotations to provide more information to static analyzers, and combining the ASA alerts with other code-related information in order to prioritize or classify alerts [13]. However, designing sound and complete tools is an intractable problem [13, 1, 31]. As a result, researchers are investing a great deal of effort in designing or optimizing alert mitigation techniques to reduce the false positive rates of static analyzers [32].

### 2.1.2 Actionable Alert Identification Techniques (AAIT)

The combination of alerts with code-related information is known as Actionable Alert Identification Techniques (AAIT). AAIT involves the use of specialized methods to determine actionable alerts to present to developers during static analysis. For the remainder of this section, we will do a literature review that is focused on the different AAIT approaches.

#### 2.1.2.1 Alert Type Selection

Heckman and Williams define alert type as a class of defects that can be detected using a detector or bug pattern [13]. Some automated static analyzers allow programmers to select the types of alerts that they would like the analyzer to detect and report. Tools

\[^1\text{http://seclab.cs.ucdavis.edu/projects/testing/tools/its4.html}\]

\[^2\text{www.dwheeler.com/flawfinder/}\]

\[^3\text{http://rat.readthedocs.io/en/latest/}\]
based on alert type selection often generate knowledge by mining code repositories or using existing information about alert types to determine alert types to select for a specific project [13, 33]. This class of analyzers is limited in that bugs or vulnerabilities that may be in the unselected categories will be suppressed and could lead to severe system flaws.

In [34], Ogasawara et al. proposed a method for assuring software quality using static analysis tools. Using alerts from a tool, called QA $C^4$, developers examined and determined whether an indicated item should be remedied or ignored [34]. From a database of over 500 alerts, 41 were selected based on the developers’ experiences with static analysis tools as a measure to reduce the number of unactionable alerts reported by the tool. However, while this approach saw a reduction in the number of alerts by 83% [13], suppressing certain alerts based on experiences alone could limit the effectiveness of the tool. This limitation lies in the fact that certain suppressed messages could be actionable and some unsuppressed messages could be unactionable. Ogasawara et al. evaluated their research by using seven (7) programs to investigate the number of and the actionability of alerts.

### 2.1.2.2 Contextual Information

This type of AAIT allows users to select the areas of code to be analyzed by a static analyzer based on knowledge of the programming project or the limitations of the analyzer. This could be for reasons such as the nature or the age of the code in addition to certain code constructs (e.g. pointers) that could be missed by the analyzer or erroneously placed in an unactionable category [35].

In [36], Yu et al. proposed an approach that uses a fuzzy inference system (FIS) to infer selection between ASA and model checking (See Section 2.1.2.7 for more

---

$^4$A quality assurance and static analysis tool for C programming language developed by Programming Research Ltd (http://www.programmingresearch.com/qac_main.html).
information on model checking) in order to find coding defects in source code. Through experimentation, Yu et al. demonstrated that FIS performed better than ASA and model checking when these approaches are done separately. FIS was slower than ASA but faster than model checking [13, 36]. FIS has a performance overhead due to the time-consuming and tedious nature of modeling checking. However, the authors proposed techniques for reducing the performance overhead, which include parallel computing, sorting of syntax trees, singularly extracting abstract syntax trees, and loading similar rules together [13]. Further, while the combined approach was able to identify more seeded vulnerabilities than the individual parts alone, no report was given on the actionability of the generated reports.

Using contextual information about the source code under analysis, Xiao and Pham designed an external extension for a tool known as Source Code Analysis System (SCAS), in order to increase the engineering efficiency of defect reporting [35, 13]. SCAS was designed to perform static analysis in enterprise software engineering environments. By observing the warning types on which developers focused most of their attention, the authors selected the top three (3) warning types, namely, missing break, unreachable code, and memory leak, as areas of focus for noise or unactionable alert reduction. They then added three (3) extensions to the tool, which include a new scan and filtering module, an updated distributed defect tracking system (DDTS), and a Pmatch tool that extends the test coverage for catching the new function mismatch problem [35]. Using this extension approach, the authors were able to suppress 33% of generated alerts [13].

2.1.2.3 Data Fusion

Data fusion is “a process dealing with the association, correlation, and combination of data and information from single and/or multiple sources to achieve improved accuracies and more specific inferences than could be achieved by the use of a single sensor alone”
In this category of AAIT, alert data is combined from multiple static analyzers in an effort to provide a more comprehensive report and to add confidence to an alert.

In 2007, Kong et al. proposed a scalable source code analysis system, known as ISA, that detects vulnerabilities using data fusion techniques [39]. The model integrates existing static analysis tools, parses the alerts and aggregates the result of the different analyses [39]. This technique allows for each tool to participate in rating an alert and for a score to be assigned. The score is used to determine the likelihood that an alert represents an actual vulnerability and helps to achieve better overall performance [39, 13]. Meng et al. also used a similar approach to merge results from multiple static analysis tools and a defect pattern database in order to display results in a universal manner. The main difference between Kong et al.’s and Meng et al.’s architecture is that Meng et al.’s architecture includes data from a defect pattern database to help in the discovery of a certain defect while the other excludes this step.

More recently (2016), Alenezi and Javed proposed Developer Companion, a framework to help developers create secure web applications [41]. The goal of the tool is to assist developers in finding security problems in web applications during implementation. While code is being written, the proposed framework uses data fusion to combine results from several static analysis tools. The code is then cross-referenced against the Common Weakness Enumeration (CWE) and National Vulnerability Database (NVD), and finally a recommendation based on the collected data is presented to the user [41]. This approach is not significantly different from Kong et al.’s ISA system except for the inclusion of data from CWE and NVD databases. Additionally, the main limitation of this framework is that it is geared toward web applications and no report is given on the actionability of the reported alerts.

2.1.2.4 Graph Theory

Graph theory is concerned with various types of networks or models of networks called graphs, which model relationships between objects [42]. An algorithm can be modeled
using a program dependence graph (PDG) to capture the dependences between the statements within objects [43]. PDGs can be combined using system dependence graphs (SDGs) to model inter-procedural dependences [43]. Since SDGs provide both the control and data flow for a program, they can be used to calculate the execution likelihood that a certain portion of code contains a static analysis alert [13]. In addition, source code repository history can be expressed as graphs and further used to capture relationships between refactored portions of code.

In [44], Boogerd and Moonen assessed the relationship between deviations from coding standard rules and actual faults, proposed two methods to quantify this relationship, and presented a technique for evaluating the actionable alert rate for alert types associated with static analyzers [44, 13]. The authors used software version history graphs, constructed from a Software Configuration Management (SCM) system, in addition to rule violations to quantify the relation between rule violations and actual faults [44]. A counter is incremented if an alert has been addressed due to fixing a fault and the remaining unaddressed alerts contribute to the overall count of generated alerts to provide a ratio or alert rate for the static analyzer in question. While the alert rate provides a prioritization measure of which alerts to address in future versions of software, drastic code changes could result in faulty scores that mislead developers, and the technique may not generalize well with other projects.

2.1.2.5 Machine Learning

Machine learning is defined “as an automated process that extracts patterns from data” [45]. Machine learning algorithms provide a mechanism for generating or learning a model that captures the relationship(s) between the descriptive attributes and the target attributes in a dataset. The learned model can be used by static analyzers to predict or prioritize the actionability of alerts by using information about the alerts as well as other information in the source code [13, 32].
Heckman and Williams proposed a process for building false positive mitigation models using machine learning algorithms (rule-based, decision tree, linear function, k-nearest neighbor, and Bayesian networks) to classify static analysis alerts as actionable or unactionable in order to reduce the number of unactionable alerts requiring review or inspection by a developer [32]. To collect data for the classification model, the authors used a four-step process, which consist of the following: 1) generate revision history from code repositories (e.g. SVN\(^5\)); 2) build each project checked out from the repositories; 3) use an analysis program to find and classify actionable alerts; and 4) generate *alert characteristics*\(^6\) for each alert. Finally, using ten-fold cross validation, the authors evaluated their approach on 15 machine learning algorithms. Their results suggest that false positive mitigation models should be project-specific [32]. While this contribution is novel, it could be time-consuming to test each model to find the best one to apply to certain projects. In addition, the authors stated that further work is required to determine if the models were overfitting the data and whether they generalize well to predict future alerts.

After observing that, for any program, alerts often follow similar patterns, Hanam et al. proposed a novel approach that uses features and alert characteristics extracted from source code surrounding the alert to predict the actionability of alerts [46]. The authors extracted alert characteristics from the site of each static analysis alert, created a feature vector, and combined this information with a priori knowledge about code patterns to compute a rank for the alerts. They evaluated their approach using *FindBugs*\(^7\) alerts that were collected from 3 Apache programs [46]. After ranking

---

\(^5\)http://subversion.tigris.org/

\(^6\)Attributes associated with an alert generated by a static analyzer and are used to predict what static analysis alerts are actionable or unactionable [46]

\(^7\)http://findbugs.sourceforge.net/
the alerts and selecting the top 5% alerts, Hanam et al. were able to classify 57 alerts as actionable while *FindBugs* alone ranked 19 actionable alerts [32]. The main difference between Hanam et al.’s technique and [32] is that Hanam et al. included alert characteristics derived from static analysis instead of deriving the characteristics solely from the source code.

In 2013, Yüksel and Sözer also applied machine learning techniques to the classification of alerts based on a set of artifact characteristics (such as severity, alert code, alert state, file name, method name, and line number) [47]. They evaluated their approach on a large commercial digital TV software using 34 machine learning algorithms from the Waikato Environment for Knowledge Analysis (Weka)\(^8\), and concluded that machine learning can be a viable approach for automatic alert classification. The main difference between this approach and [32] is that Yüksel and Sözer used several snapshots from the alert history to create the training/test sets and conducted their experiments using an industrial software.

### 2.1.2.6 Dynamic Detection

While dynamic analyzers fall into a different category of security testing tools (See Section 2.2), the results of these tools can be used to drive or improve static analysis. Dynamic analysis, which involves analyzing an application during execution, requires the use of test cases as input in order to detect coding defects. Because the supplied test cases may cause the system to result in faulty behavior [13], the location of the areas of code that trigger errors can be used in static analyzers to improve the actionability of alerts. Static analysis techniques can also be used to improve the generation of test cases for dynamic analyzers [48]. Further, dynamic detection may utilize select code to identify faulty program execution [13].

\(^8\)https://www.cs.waikato.ac.nz/ml/weka/
Qing et al. implemented a tool called \textit{IntFinder} for finding integer bugs in C/C++ program code based on a combination of static and dynamic analysis \cite{49}. After decompiling x86 binary code and creating the suspect instruction set through a combination of dynamic analysis with taint analysis, \textit{IntFinder} dynamically inspects the instructions in the suspect set to determine the instructions that are actual integer bugs \cite{49}. First, a decompiler is used to translate compiled code into static single assignment (SSA) constructs. Then, an extended type analysis is done based on exploited functions and statements to extract more information from the binary code to create the suspect set \cite{49}. The authors studied 350 integer bugs in the CWE database and selected the top four, which are associated with memory allocation function, array index, memory copy function, and signed upper bound check. The extension of the type analysis coupled with dynamic and taint analysis has allowed the authors to find integer bugs beyond integer overflow, which is common in the literature \cite{50, 51}. The main limitation of this approach is the overhead and incompleteness of decompilation because not all functions are decompiled. Additionally, generating test cases for dynamic analyses can result in high false positive and false negative rates, especially since test case inputs are extracted from known vulnerabilities (i.e. it will not perform well on unspecified vulnerabilities).

In addition, Csallner et al. proposed a tool called \textit{DSD-Crasher} that finds bugs without displaying false alerts based on a three-step approach: \textbf{D}: dynamic inference, \textbf{S}: static analysis, and \textbf{D}: dynamic verification \cite{48}. The authors first used the \textit{DIAKON} tool \cite{52} to infer likely program invariants from an existing test suite and then feed the output into their intermediate \textit{Check ‘n ‘Crash} tool, which incorporates \textit{ESC/Java} static analysis \cite{53, 54} to output locations for potential Java runtime exceptions \cite{48, 13}. \textit{Check ‘n ‘Crash} then compiles code that satisfy certain error conditions into applicable test cases that are executed with \textit{JCrasher} \cite{55} to determine whether the error is reproducible given a specific input. Csallner et al. compared their approach with existing dynamic detection approaches and concluded that the main improvement they have made over existing approaches is the inclusion of a static analysis phase that “exhaustively attempts
to explore program paths and yield a directed search through the test space”[48]. This results in \textit{DSD-Crasher} being able to better detect bugs and limit false bug warnings [48]. In addition to the fact that \textit{DSD-Crasher} only identifies runtime exceptions, one major limitation is based on its inclusion of \textit{DIAKON}, which requires regression tests to provide full coverage [13].

[56] proposed a classifier that reduces manual program validation by dynamically classifying statically-generated memory leak warnings in C/C++ programs as MUST-LEAK, LIKELY-NOT-LEAK, BLOAT, and MAY-LEAK. The authors used coding patterns to direct \textit{Fortify}\(^9\) to report false warnings which enabled them to create four classes of warnings. The proposed method uses allocation site \textit{a}, CFG fragments, and a function return point \textit{p} to direct a concolic testing engine\(^10\) to generate test cases that cover \textit{p} and then dynamically track each object created by \textit{a} in the test runs to validate whether leaks occur in the program under test [56]. The main limitation of this approach is that it only classifies memory leaks.

### 2.1.2.7 Model Checking

Model checking is “a verification technique that explores all possible system states in a brute-force manner”[57]. Since model checking can be used to verify that a model of a program meets certain requirements, it is considered a sound and complete verification technique [13]. However, model checking has its limitations, which are due to the explosive and expensive nature of checking all abstractions of a model to ensure that the model satisfies certain properties. Still, static analyzers can use alerts to focus model checking to certain areas of a program — a technique known as directed model checking.

\(^9\)http://www.ndm.net/sast/hp-fortify-static-code-analyzer

\(^{10}\)modification of the CREST tool located at http://code.google.com/p/crest
In [58], Rungta and Mercer presented a metaheuristic with the goal of reducing the manual effort required to verify warnings generated by static analyzers. A metaheuristic is “defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space; learning strategies are used to structure information in order to find efficiently near-optimal solutions” [59]. The authors manually generated locations in code that represent points of interest and controlled scheduling decisions in a model checker using metaheuristic in order to check the feasibility of a particular static analysis warning [58]. They evaluated their approach via a case study on the JDK 1.4 library, which discovered concurrency errors [58]. The main limitation of this approach is that it requires manual generation of the input sequences used to direct the execution of the program and model checker.

2.1.3 Clustering

Clustering involves grouping objects according to a certain similarity metric. Clustering can be used to group alerts in order to reduce the number of alerts reported to the user during static analysis. Lee et al. proposed two algorithms that reduce the manual effort required to investigate static alarms by clustering alarms according to their sound dependence information [60]. “An alarm $A$ is said to have sound dependence on alarm $B$ if alarm $B$ turns out to be false, then so does alarm $A$ as a logical consequence.” [60] The first algorithm finds the minimal dominant alarms from a set of general alarms and a set of groupable alarms, and the second algorithm finds a set of non-minimal alarms based on a CFG where every node has predecessors and successors specified by a certain function. The authors argued that their proposed framework is applicable to any semantics-based static analyzer. However, the framework was only applied to buffer-overflow detection on fourteen (14) packages from begbunch [61], GNU and Sourceforge programs.
Moreover, in [62], Muske et al. proposed partitioning techniques to identify redundant warnings produced during static analysis in an effort to reduce review efforts required by developers. Unlike AAIT approaches that attempt to group alerts as actionable or unactionable, [62] considered all warnings as actionable. They employed a data flow analysis technique, based on a CFG, to group related warnings by choosing a leader and marking the related warnings as redundant (or follower warnings). This approach is time-saving as it decreases the number of alerts that a developer is required to review.

Similarly, Zhang et al. presented a sound alarm correlation algorithm that automatically computes correlations based on forward analysis, trace partitioning, state slice and semantic slice, and uses this information to reduce the number of alarms reported to the user [63]. They applied the technique to the null-pointer dereference bug using their own in-house static analyzer, known as DTSGCC, where they saw a reduction of 33.1% in the number of alarms reported.

Fry and Weimer introduced a parametric technique to cluster machine-generated defect reports using similarity metrics so that similar bugs can be triaged and potentially fixed in aggregate [64]. The proposed model clusters alarms using both syntactic and structural judgments of relatedness based on information reported by Coverty SA and FindBugs static analysis tools [64]. The authors applied similarity metrics (e.g. Levenshtein edit distance, punctuation edit distance, etc.) to all pairs of applicable report sub-components to obtain similarities for each pair of reports [64]. A classifier is used to examine the report pairs for similarity based on a linear combination of weighted feature values. A two-phase clustering process is used to group defects. First, an undirected graph of similarity scores and unclustered defects is constructed. Next, the maximum clique is found and the corresponding defects are removed and clustered recursively. While the technique resulted in better performance than code clone tools

\[11\text{https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html}\]
such as ConQAT, PMD and Checkstyle, its reliance on Coverity SA and FindBugs may not generalize to all static bug finders.

2.2 DAST Approaches

In Section 2.1.2.6, we discussed the combination of dynamic analysis with static analysis in an effort to improve the actionability to static alerts. Now, we will look at dynamic analysis as a category of security testing. Incorporating security in the SDLC should not be a one time approach. Therefore, researchers have proposed a number of models that include both DAST and SAST. For example, Figure 2.2 [2] shows an approach to security testing where security is viewed as a process from requirements to deployment and maintenance and is expected to ensure the design of more secure programs instead of traditional SDLC.

Figure 2.2: Secure software development life cycle showing the inclusion of both SAST and DAST. [2]
In 1975, Ramamoorthy and Ho’s research was one of the first to examine dynamic analysis. The authors emphasized the challenge with doing formal proof of correctness as a means of software validation and proposed using “automated tools and software evaluation systems as a means to improve the reliability and reduce the cost of large software systems” [4]. Since that time, a great deal of effort has been invested in the identification of strategies and techniques for automated generation of test data to improve dynamic application security testing [65, 66, 67, 68]. DAST techniques can be used to detect faults in programs, but Gaudel indicated that “locating and correcting faults are generally regarded as out of the scope of testing.” [69]

2.2.1 Automated Software Test Case Generation

For the remainder of this section, we will review literature from five (5) prominent areas [68, 70] of automated software testing that are related to program security. These areas include: symbolic execution, model-based testing, combinatorial testing, adaptive random testing, and search-based testing.

2.2.1.1 Symbolic Execution

Symbolic Execution is a white-box\textsuperscript{12} program analysis technique that uses symbolic input values, instead of concrete (initialized) data, to analyze a program’s code [68]. During execution of a program being tested symbolically, one can obtain information on the symbolic values of the variables, path constraints, and the values of a program counter. As the path constraints are updated, this helps the programmer to determine if a certain path yields a solution, since the program will end if a path is unsatisfiable. If the program terminates when a certain path fails or a bug is reached, a test case can be generated by solving the current path condition for concrete values [71], which can be used to

\textsuperscript{12}the analysis of a program’s source or binary code to generate test data
reproduce the bug in a deterministic environment. While this kind of software testing has been around for decades, it has received much more attention as of late due to the proliferation of powerful computers capable of solving complex constraints. In the area of program security, [71, 72, 54] proposed the use of symbolic execution to generate test data to improve code coverage and expose software bugs. Additionally, [73] utilized symbolic execution to automatically generate security exploits. Like model checking, which was discussed in Section 2.1.2.7, symbolic execution suffers from path explosion in addition to path divergence and complex path constraints, especially on real world problems [68].

In [54], Khurshid et al. argued that software testing does not do well at finding errors in concurrent systems and proposed a novel framework, based on symbolic execution, that automatically generates test cases and performs model checking in concurrent programs. The framework is geared towards concurrent programs that accept inputs from unbounded domains and also supports complex data structures. Their proposed algorithm uses lazy initialization\(^\text{13}\) to handle uninitialized variables and method post-conditions as test oracles to check the correctness of a method [54]. In addition, it handles destructive updates that may lead to bugs by creating mappings between objects with initialized variables and those without. Khurshid et al. handled space explosion by using partial order and symmetry reductions [54]. However, no other techniques were proposed to handle the other inherent problems with model checking.

[71] describes a symbolic execution tool, KLEE, which automatically generates test data using search heuristics to achieve high code coverage. For evaluation, Cadar et al. conducted experiments by running KLEE on over 452 programs (programs within the GNU COREUTILS package, BUSYBOX and MINIX utilities). The automatically generated tests achieved a coverage of 84.5% of the lines of code in GNU COREUTILS, uncovered ten memory-based, fatal errors in COREUTILS, and achieved 90.5% of

\(^{13}\text{initializing variables on an “as-needed” basis}\)
coverage in BUSYBOX [71]. Code coverage was measured based on executable lines of code (ELOC) and calculated using gcov\textsuperscript{14}. While KLEE was built upon the LLVM\textsuperscript{15} compiler infrastructure, which supports a wide range of languages, Cadar et al. evaluated their tool on C programs (bytecode) and no mention was made of its support for other languages. Other limitations of the tool include its lack of support for symbolic floating point, threads, and the requirement that memory objects should have concrete sizes [71].

Godefroid et al. discussed some limitations of black-box testing\textsuperscript{16} and proposed a generational search algorithm in the area of white-box fuzz testing to find defects in software. The limitations of black-box testing includes poor code coverage that could be detrimental especially in the security context. The proposed algorithm works in a repetitive manner by symbolically executing a program, formulating input constraints from conditional statements encountered during execution, negating the constraints and solving them with a constraint solver to generate variants of the previous input that target different paths in the program [72]. As the search is done, a bounding parameter is used to limit the backtracking of each sub-search so that exploration of overlapping parts of the search space is minimized. This approach is expected to maximize code coverage and find defects speedily. The authors observed that a single symbolic execution task is significantly slower than testing or tracing a program, but was leveraged by the generational search [72]. They implemented the algorithm in a tool known as Scalable, Automated, Guided Execution (SAGE)). One limitation of the tool is that it focuses only on file-reading applications [72].

\textsuperscript{14}https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html

\textsuperscript{15}http://llvm.org/

\textsuperscript{16}a software testing technique that involves testing software for functionality while ignoring the internal mechanism of the source code or system [74]
In [73], Brumley et al. proposed techniques for automatic patch-based exploit generation (APEG) and showed the possibility of such generation using five (5) recently-patched Windows programs. The authors argued the fact that there is ample time between patch distribution and the time that unique IPs check for new patches where attackers who received the first patches could exploit vulnerabilities. They observed that input-validation bugs are usually mitigated by hardening sanitization checks via patches; therefore, inputs that fail certain checks could be automatically generated [73] using automatic goal-based test generation techniques. Their proposed techniques are based on combinations of static and dynamic analyses. The static analysis approach generates constraint formulas from a control flow graph (CFG) and then focused on multiple graph paths. They used a dynamic approach to generate constraint formulas from a sample execution. The process works by first using EBDS (eEye’s Binary Diffing Suite)\(^\text{17}\) to compare and report the differences between two binaries, filtering the resulting differences for new input validation checks, generating constraint formulas, and using STP\(^\text{18}\) to generate candidate exploits from the constraint formulas. Using these techniques, Brumley et al. showed that they could discover exploits for the Internet Group Management Protocol (IGMP), the Windows Graphic Device Interface (GDI), Portable Network Graphics (PNG), the ASPNet.Filter, and DSA.SetItem. The authors then suggested patch obfuscation and encryption and faster patch distribution as mitigation for APEG.

\(^{17}\)Open source tool for performing automated binary differential analysis located at http://www.opencrc.org/downloads/details/226/eEye_Binary_Diffing_Suite.(EBDS)

\(^{18}\)a decision procedure for quantifier-free first order logic with bit-vector and array datatypes[75]
2.2.1.2 Model-based Testing

Model-based testing (MBT) is “a variant of testing that generates test cases based on explicit behavior models that encode the intended behaviors of a system under test (SUT) and/or the behavior of its environment” [76]. There are three main approaches in MBT, namely, axiomatic, finite state machines (FSM), and labeled transition systems (LTS) [68]. We will define and explore existing security-related work in these areas below.

Axiomatic approaches

Axiomatic approaches utilize logic calculus (and by extension, algebraic specification) to outline rules for testing a system. For example, Gaudel created a framework for software testing based on algebraic specification [69]. Anand et al. stated that “the algebraic approach can only test a single function or sequence, and in its pure form it is not of practical relevance today” [68]. Later in 2013, Brucker and Wolff built upon the foundations of higher-order logic to create a framework by combining theorem-proving and testing to improve the Isabelle\(^\text{19}\) theorem prover to provide automatic procedures for test case generation and test data selection [77, 68].

Finite state machine (FSM)

A finite state machine (also known as finite automata or Mealey machine) is “a model that describes the dynamic behaviors of an object over time” [78]. In this model, inputs and outputs are paired during transitions and input sequences are derived from the machine based on a certain coverage criteria [68]. Since software is always in a certain state, the current state governs what set of inputs testers can supply to the system [79], making FSMs a good fit for modeling software states logically. However, FSMs are not expressive enough to model real software systems [68, 79]. As a result, extended

\(^{19}\)https://www.cl.cam.ac.uk/research/hvg/Isabelle/
finite state machines (EFSMs) are used to augment FSMs with state variables and data parameters for inputs and outputs, so they can be applied to real-world software testing [68].

Keum et al. presented a procedure for extending WSDL (Web Services Description Language) using EFSMs in order to generate test cases that yield significant code coverage and reveal potential faults in web services. The authors argued that existing web service testing methods are syntactic because WSDL does not capture dynamic and behavioral information [80]. While there are similarities between communication protocol testing and the statefulness of web services, WSDL does not contain sufficient information for testing web services [80]. Keum et al. resolved this limitation by adding temporal ordering information to WSDL to describe the behavior of web services and make them easier to model using EFSMs [80]. However, this approach still has its limitations especially for compositions of web services due to the inherent overhead of using EFSMs to generate test cases.

*Labeled transition systems (LTS)*

[81] defined labeled transition system as:

... a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

Tretmans noted that while “labelled transition systems constitute a powerful semantic model to reason about processes, such as tests” [81], realistic systems have a large number of states and may require a process language to represent them using behavior expressions.

Cartaxo et al. presented a model-based testing procedure for functional test case generation to help with feature testing in applications on mobile phones [82]. The procedure involves transforming sequence diagrams into an LTS model that captures
use scenarios (actions executed and viewed by the user), identifying all paths in the LTS using a depth first search (DFS), and tabulating the paths to be used as test cases [82]. The authors tested the procedure on the “Go to URL” functionality present in embedded items in text messages, from which they were able to generate two (2) test cases. However, further application of the procedure is required to measure its effectiveness.

2.2.1.3 Combinatorial Testing

Combinatorial testing is “a specification-based testing criterion, which requires that for each $t$-way combination of input parameters of a system, every combination of valid values of these $t$ parameters be covered by at least one test case” [83]. Shiba et al. reported that the use of a test set that covers all $t$-way combinations of the parameter values (or configuration settings) for a small integer $t$ can be very effective in testing a system, since faults usually present themselves during interactions between a few parameters [83]. The most common type of combinatorial testing is combinatorial interaction testing (CIT). CIT involves including all $t$-way combinations of parameter values in a sample of inputs and testing only the selected configurations [68, 84]. The majority of research [85, 86, 87, 88, 89] in this area has been focused on discovering new techniques to generate CIT samples, which are based on heuristics [68] and are often improvements over the AETG (Automatic Efficient Test Generator) algorithm [85]. AETG uses a greedy strategy to construct a test set by repeatedly adding a test case that covers non-covered interactions between parameters or configurations [83]. For example, Shiba et al. proposed a modification to AETG by using a genetic algorithm (GA) and an ant colony algorithm (ACA) to generate test data. However, no notable improvements in test set quality or practical advantages were noted between GA, ACA and AETG.

[90] is one of the more recent applications of combinatorial testing to the area of software security. Simos et al. summarized research done by the National Institute of
Standards and Technology (NIST), which focused on parsing untrusted web content, web application security, system call testing, hardware trojan detection, and protocol interaction testing. The researchers compared the cumulative percentage of faults at $t = 1-6$ in six (6) applications, and proposed the establishment of a new research field known as *combinatorial security testing*. Their results showed that “fault detection rate increases rapidly with interaction strength, up to $t = 4$” [90], capping off at $t = 4-6$, thereby confirming that it is challenging to detect faults in software for lower values of $t$; therefore, testing should be focused on interactions between few variables. Further, the authors coordinated with the World Wide Web Consortium (W3C) to use NIST Automated Combinatorial Testing for Software (ACTS) tool to create test cases that resulted in 100% coverage of the DOM’s parameter interactions for $t = 2$ and $t = 3$. This effort revealed a vulnerability in cross-site scripting (XSS), which was later fixed by W3C. Additionally, using combinatorial testing, the authors discovered over 50 cases of XSS vulnerabilities within the Koha integrated library management system\footnote{https://koha-community.org} and numerous errors in Linux kernel, reduced the test suites’ size for Hardware Trojan activation, and presented a coverage measurement for recommendations on available TLS cipher suites, for various $t$ values [90].

### 2.2.1.4 Adaptive Random Testing

Adaptive random testing (ART) is a proposed improvement to random testing (testing a program with normally distributed test cases). It involves distributing test cases more evenly within the input space [91]. Adaptive random testing came about as a result of two observations: 1) the use of failure rates\footnote{the ratio of the number of failure-causing inputs to the number of the set of all possible inputs [68]} alone to measure the effectiveness of a random test is insufficient [91]. 2) software failures result from faults in contiguous
regions of the input domain\cite{92, 91, 68}. Chan et al. noted that, in addition to failure rates, partition-based testing strategies should also be measured by the geometric pattern of the inputs that cause failures \cite{93}. To address these concerns, Chen et al. proposed using F-measure\textsuperscript{22} as a measure of effectiveness of random testing, and evenly spreading test cases over the input domain \cite{91}. From our research, the application of adaptive random testing to program security has not been widely explored. Most of the research in adaptive random testing has been focused on improving failure-finding efficiency and the distance metrics used to measure the distance between two test cases \cite{92, 94, 95}. Researchers \cite{68, 96} described Chen et al.’s approach as a near-optimal, lightweight approach. However, while ART outperforms RT with respect to the F-measure and P-measure, ART algorithms are expected to have higher space and time complexities than random testing due to the additional task of evenly distributing the test cases \cite{91, 68}. Anti-random testing also espouses spreading test cases evenly within the input space, but differs from ART in that anti-random testing is deterministic and requires that testers choose test cases in advance \cite{68, 97}.

\textbf{2.2.1.5 Search-Based Testing}

Search-based software testing (SBST) is “a branch of search based software engineering (SBSE), in which search algorithms are used to automate the process of finding test data that maximize the achievement of test goals, while minimizing testing costs” \cite{68}. Harman et al. observed that “59\% of the overall SBSE literature are concerned with software engineering applications related to testing” \cite{66}. The main focus of SBST is to use search-based algorithms to generate test cases that fulfill a testing objective. These algorithms are guided by a fitness function, and much of the SBSE research has been focused on tuning or optimizing the fitness function \cite{98, 66, 68}. Common

\textsuperscript{22}the expected number of test cases required to detect the first failure
optimization approaches include hill climbing [99], simulated annealing [100], and genetic algorithms [101]. Current research in SBST is centered on the Oracle Problem\(^{23}\) as well as formulating hybrid solutions to the test generation problem by combining SBST with other techniques, such as dynamic symbolic execution [103, 68]. As noted in Section 2.2.1.1, symbolic execution techniques do not handle floating point constraints well, due to the limitations of constraint solvers, so hybridizing dynamic symbolic execution with SBST has been proposed as a solution.

[70] proposed a dynamic test data generation framework based on genetic algorithms. The framework consists of a basic program analyzer system (BPAS) and a test case generator, which supports static analysis, CFG creation, and test case evaluation. The Automatic Test Cases Generation System (ATCGS) is built upon two algorithms: the Batch-Optimistic (BO) and the Close-Up (CU) algorithms. The BO algorithm utilizes a genetic algorithm to evolve sets of test cases and converge to a near-optimum set based on a edge/condition fitness function [70]. The CU algorithm uses a program control flow graph to determine unreachable elements (targets) that were not covered by the BO algorithm [70]. The authors compared their approach to four other testing algorithms: random testing, gradient descent, standard genetic algorithm, and the differential genetic algorithm. Their framework achieved 100% edge/condition coverage regardless of the program under testing, unlike the other approaches [70]. The main limitation reported by the authors is that the use of a control flow graph limits the program to being able to depict only method-based features (ie. inability to describe class-based features), which further limits the code coverage system to what is captured by the control flow graph [70].

\(^{23}\)distinguishing the corresponding desired, correct behavior from potentially incorrect behavior [102]
2.3 Hybrid Analysis

Hybrid analysis involves combining output from dynamic analyzers with the output from static analyzers in order to achieve better code coverage and vulnerability detection.

In 2006, Aggarwal and Jalote presented a model which handles buffer overflow vulnerability by employing both static and dynamic analysis as it relates to the `strcpy` library function in C program code [14, 13]. They combined static and dynamic analyses to identify areas of code that require buffer overflow analysis. One of the disadvantages of dynamic analysis is the requirement of a large number of test cases, which present an overhead. While the authors were able to reduce the number of test cases required by combining both static and dynamic approaches, performance may still be impacted since there is a requirement to mark the locations of `strcpy` functions [13]. Further, two other limitations with this approach are that buffer overflow is not restricted to the `strcpy` function and the tool only supports the C programming language. As a result, only a subset of buffer overflows may be detected by this approach.

In 2012, [15] proposed a hybrid approach that uses source code program slicing to reduce the size of C programs while performing analysis and test generation. The authors built upon the static analysis and testing (SANTE) method, which performs static analysis and dynamic analysis on full programs, by using a minimal slicing-induced cover and alarm dependencies to diminish the costly calls of dynamic analysis. They implemented their program using the FRAMA-C\textsuperscript{24} framework and PATHCRAWLER\textsuperscript{25}, evaluated the system using five sub-programs that contained out-of-bound and invalid pointer bugs, and proved that using SANTE with program slicing is more efficient than with full programs.

\textsuperscript{24}an open-source extensible and collaborative platform for static analysis of C programs located at http://frama-c.com/

\textsuperscript{25}prototype tool for automatic test-case generation [104]
In 2014, McCorkendale et al. achieved a patent for systems and methods for combining static and dynamic code analysis. The idea is to use information from both static and dynamic analyses to determine whether executable code leaks sensitive data via one or more objects [105]. The proposed patent describes the traditional combination of static and dynamic approaches, which have been discussed in the literature [106, 107] and are reportedly unsuccessful due to the time-consuming nature and the signal-to-noise ratio.

In 2015, [16] applied a program slicing technique, similar to [15], to create a tool called Flinder-SCA. The authors also implemented their program using the Frama-C platform. The main difference between [15] and [16] is that [16] uses Search Lab’s Flinder testing tool and performs abstract interpretation and taint analysis to detect alarms and tries to confirm detected alarms by fuzzing the reduced program [16] whereas [15] does not perform taint analysis or fuzzing. They evaluated their work by applying it to the Heartbleed bug. The main limitation of this approach is that the user is required to specify the potentially tainted inputs and the tool needs improvement in order to detect other vulnerabilities [16].

2.4 Related Work

We now provide a summary of more recent works that are more closely related to the ideas that comprise this research.

2.4.1 Mining topic models from source code

Gopalakrishnan et al. [108] presented a bottom-up approach to software design that recommends architectural tactics (a quality-attribute-response) based on topics discovered from source code projects. They used a classifier in addition to a recommender system to predict where tactics should be placed in a programming project to improve the quality, but not security, of the code.
2.4.2 Machine Learning/AI Systems

In [109], Medeiros et al. presented the DEKANT tool that automatically detects web-based vulnerabilities using hidden markov models (HMM). First, the tool extracts code slices from source code and translates these slices into an intermediate slice language (ISL). It then analyses the representation to determine the presence of vulnerabilities in code written in PHP.

In [110], the authors described a tool, known as HACKAR, that uses an improved version of Java PathFinder (JPF) to execute Java programs and identify vulnerabilities. The tool is a dynamic analyzer that formulates a problem using Satisfiability Modulo Theory (SMT) and uses symbolic execution to determine program paths that may lead to vulnerabilities. In addition, HACKAR uses Artificial Intelligence (AI) to learn the semantics of tasks based on program traces in order to produce a knowledge base for providing advice to programmers on how to fix vulnerabilities.

2.4.3 Code Completion

In [111], Raychev et al. presented an approach to code completion based on a novel combination of program analysis with statistical language models. Given a codebase, their system first extracts abstract histories in the form of sentences from the data. Then, these sentences are fed to a language model such as an n-gram model or recurrent neural network model that learns probabilities for each sentence.

Also, in [112], the authors described an architecture that allows library developers to introduce interactive and highly-specialized code generation interfaces, called palettes, directly into the editor. Both of these code completion approaches are based on system design and sentence suggestion and have not been applied to vulnerability detection and mitigation.
2.4.4 Difference Between our Approach and Existing Approaches

As discussed before, this work couples machine learning algorithms with static analysis to analyze program code as the programmer types, compares the user’s code with a knowledge base of unsafe practices to determine the presence of unsafe code and recommends fixes by providing example code to the programmer during development. While [110] uses AI to learn about the user program, it requires that the program be symbolically executed in order to find vulnerabilities. As discussed in the literature [68], symbolic execution suffers from path explosion, path divergence and challenges with complex path constraints, especially on real world problems. This presents challenges with the generalizability of the solution as confirmed by the authors [110].

In [109], an intermediate language is required to annotate tainted functions in the code. In contrast, the proposed model in this research works directly with the parse tree of the source code to detect patterns for automatic detection and classification of vulnerabilities. Further, this work mines a large code base and uses the safe examples to provide not only advice but also fixes to the programmer.
Chapter 3

Proposed Approach

This chapter describes the approach proposed, implemented, and evaluated in this research. The proposed approach is a recommender system that uses a classification system to recommend fixes for potential vulnerabilities in program code.

3.1 Overview of Approach

The approach consists of two main phases (modeling and application) and three main components (the data analyzer, the classification system, and the recommender system) as shown in Figure 3.1. Here, we briefly describe each component. Chapter 6 discusses the modeling phase in more detail while Chapter 7 covers the application phase (system design and implementation).

The first phase in the proposed approach is the modeling phase. This phase involves analyzing data collected from the National Vulnerability Database (NVD) in addition to open source programs to identify features for detecting a set of vulnerabilities. These features are then used by a data analyzer to process program code and create feature sets for training classifiers in the classification system to detect each vulnerability.
The second phase involves feeding a user program into the recommender system that is connected to the classification system, which in turn classifies the program code based on the knowledge of the classifiers and outputs recommendations that include examples for fixing each vulnerability.

### 3.2 The Data Analyzer

The data analyzer consists of hand-coded feature extractors that are designed based on vulnerability descriptions and fixes from the NVD. The analyzer accepts as input open source program code and outputs feature sets for detecting a set of vulnerabilities. We propose mining open source projects and categorizing source code in order to train classifiers for detecting and mitigating each vulnerability. Effectively training classifiers requires sufficient data. Therefore, we propose the use of a distributed framework such as MapReduce to extract features from a large collection of code repositories.

### 3.3 The Classification System

This component uses classifiers that are trained with labeled data from the data analyzer to classify source code as safe or unsafe. Several classifiers will be evaluated to determine the most appropriate one to be employed in the final system. The output of the
classification system is a model that can classify each vulnerability. The labeled datasets used to train the classifiers will also provide safe code examples to be used by the recommender system to recommend fixes to the programmer.

### 3.4 The Recommender System

The recommender system incorporates the models that are trained by the classification system. It accepts the user’s code and utilizes the data analyzer to create a feature set/data object from the given program code. The classifiers then determine the most appropriate classification for the data object. If the data object is unsafe, a predefined recommendation that includes a warning that contains a list of unsafe method(s) and variables found in the user’s code are displayed to the user. The recommendation will also include ranked fixes for each vulnerability. Fixes will be ranked using text similarity schemes in order to display a list of examples that resemble the code being developed. We propose the use of IntelliSense to initiate the recommender system as the programmer types in order to help the programmer mitigate potential vulnerabilities as soon as possible. IntelliSense, also known as code-completion or code-hinting, refers to productivity features that help programmers learn about their code by keeping track of parameters and providing the ability to add properties to code during development. We conduct formative testing to determine the utility of augmenting static analysis with this approach in helping programmers write more secure code in Chapter 7.
Chapter 4

Data Understanding

This chapter describes the data that is used in this research. First, we describe the data obtained from the National Vulnerability Database followed by a description of the Sourcerer 2011 dataset, which is used for feature extraction and model building.

4.1 The NVD/CVE

Launched in 1999, the Common Vulnerabilities and Exposures (CVE) is a dictionary of common identifiers for publicly known cyber security vulnerabilities, which is hosted by the MITRE Corporation[113]. The MITRE Corporation maintains CVE and the CVE public website, oversees the CVE Numbering Authority (CNA) and CVE Board, and provides technical guidance to ensure CVE serves the public interest. A CVE entry is created after a vulnerability submission is made by a responsible and trusted party. Each entry is reviewed by a team of experts and is assigned a unique identifier (CVE ID) by a CNA, a description, and references. Each reference identifies the source of the vulnerability, notes the associated CVE ID, and uses a well-defined identifier to facilitate searching a source’s website. A CVE entry may be marked as “DISPUTED”
during disagreements between parties, “REJECT” when it is not accepted as a valid entry, “RESERVED” when it is being processed by the CNA or security researcher, or “ACCEPT” after it satisfies requirements.

The U.S. National Vulnerability Database (NVD) is a “comprehensive cybersecurity vulnerability database that integrates all publicly available U.S. Government vulnerability resources and provides references to industry resources”[113]. NVD is provided by the National Institute of Standards and Technology (NIST). NVD enhances the information in CVE to deliver more details for each CVE entry such as fix information, severity scores, and impact ratings according to a Common Vulnerability Scoring System (CVSS)[114]. CVE is related to the Common Weakness Enumeration (CWE) in that CVE entries in the NVD database are classified according to the Common Weakness Enumeration classification system and many CWE vulnerabilities are also included in CVE[115]. NIST releases the NVD database as data feeds in a number of formats such as XML, JSON, and RSS. The filenames follow the format nvdcve-1.0-yyyy.json, where “yyyy” represents the year in which a set of vulnerabilities were added to the database. Figure 4.1 shows the schema for the XML 2.0 version of the NVD database.

4.2 The Sourcerer 2011 Dataset

For our work, we used the Sourcerer 2011\(^1\) dataset, which is provided by the University of California - Irvine. This dataset is a collection of artifacts based on over 70,000 Java projects and approximately 100,000 jar files that were collected from Apache, Google Code and Sourceforge in 2011 [116]. Figure 4.2 shows the structure of the repository. As can be seen from the figure, the dataset is divided into four (4) TAR archives, identified as aa to ad. Each of these archives contains varying numbers of projects,

\(^1\)http://sourcerer.ics.uci.edu/
This schema defines the structure of the National Vulnerability Database XML feed files version: 2.0. The elements and attribute in this document are described by xsd:annotation tags. This file is kept at https://scap.nist.gov/schema/nvd/nvd-cve-feed_2.0.xsd. The NVD XML feeds are available at http://nvd.nist.gov/download.cfm.

Release Notes:
Version 2.1: * Modified to use new CVE-ID format
Version 2.0: * Redesign of the feed to integrate with the new vulnerability data model schema.
Version 1.2: * CVSS version 2 scores and vectors have been added. Please see http://nvd.nist.gov/cvss.cfm?vectorinfo and http://www.first.org/cvss/cvss-guide.html for more information on how to interpret this data.

Figure 4.1: NVD XML 2.0 Schema

which are numbered in a sequential manner. Each project is then organized into a cache of important files, the content, which follows the organization system used by the developers, and a project.properties file, which contains information such as the repo URL and author. This dataset is ideal because it was aggregated the same year the top
25 CWE were released. Therefore, we expect to find some of these vulnerabilities in these projects.

4.3 Data Collection

We obtained access to the CVE data feeds for 2017 in XML format (nvdcve-2.0-2017.xml). In addition, we also collected data regarding the top 25 SANS/CWE, which are also a part of the CVE dataset. Table 4.1 shows the top 25 CWE. We utilized the top 2 CWE/CVE from this list in model building and evaluation in Chapters 6 and 8.
Table 4.1: 2011 CWE/SANS Top 25 Most Dangerous Software Errors [3]

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>2</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>3</td>
<td>79</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>4</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>5</td>
<td>76.9</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
</tr>
<tr>
<td>6</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>7</td>
<td>75</td>
<td>CWE-786</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>8</td>
<td>75</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>9</td>
<td>74</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>10</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>11</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>12</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgeries (CSRF)</td>
</tr>
<tr>
<td>13</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
</tr>
<tr>
<td>14</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>15</td>
<td>67.8</td>
<td>CWE-863</td>
<td>Incorrect Authorization</td>
</tr>
<tr>
<td>16</td>
<td>66</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
<tr>
<td>17</td>
<td>65.5</td>
<td>CWE-732</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
</tr>
<tr>
<td>18</td>
<td>64.6</td>
<td>CWE-676</td>
<td>Use of Potentially Dangerous Function</td>
</tr>
<tr>
<td>19</td>
<td>64.1</td>
<td>CWE-327</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
</tr>
<tr>
<td>20</td>
<td>62.4</td>
<td>CWE-131</td>
<td>Incorrect Calculation of Buffer Size</td>
</tr>
<tr>
<td>21</td>
<td>61.5</td>
<td>CWE-307</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
</tr>
<tr>
<td>22</td>
<td>61.1</td>
<td>CWE-601</td>
<td>URL Redirection to Untrusted Site ('Open Redirect')</td>
</tr>
<tr>
<td>23</td>
<td>61</td>
<td>CWE-134</td>
<td>Uncontrolled Format String</td>
</tr>
<tr>
<td>24</td>
<td>60.3</td>
<td>CWE-190</td>
<td>Integer Overflow or Wraparound</td>
</tr>
<tr>
<td>25</td>
<td>59.9</td>
<td>CWE-759</td>
<td>Use of a One-Way Hash without a Salt</td>
</tr>
</tbody>
</table>

These two vulnerabilities were chosen because of their severity scores in the CWE list and their frequency in NVD. Figure 4.3 shows that more than 1,500 vulnerabilities in the CVE 2017 list were caused by the top 10 SANS/CWE of 2011.
Figure 4.3: Number of vulnerabilities in the NVD 2017 List that were caused by the top 10 SANS/CWE of 2011. The plot also shows the CWE severity score for each CWE.
Chapter 5

Data Analyzer Environment
Setup

In this chapter, we describe the steps followed to set up the environment for processing the data.

5.1 Installing and Configuring Apache Hadoop for Running MapReduce Tasks

In our work, we used Apache Hadoop as a MapReduce environment to process the Sourcerer dataset. MapReduce is a programming model and an associated implementation for processing and generating large datasets [117]. The concept behind MapReduce is the specification of computations in terms of a map and a reduce function that can be executed in parallel across large-scale clusters of machines. The map and reduce functions interact with key value pairs as follows:
map\((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)

reduce : \(\text{list}(k_2, v_2) \rightarrow \text{list}(k_3, v_3)\)

The main goal is to distribute tasks across multiple nodes in order to decrease the effect of machine failures and schedule inter-machine communication to make efficient use of the network and disks [117].

The Apache Hadoop software library is one of the most popular implementations of the MapReduce methodology. It is a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model [118]. The Apache Hadoop 3.0 release line contains the following modules and utilities [119]:

- **Hadoop Common**: The common utilities that support the other Hadoop modules.

- **Hadoop Distributed File System (HDFS\textsuperscript{TM})**: A distributed file system that provides high-throughput access to application data.

- **Hadoop YARN (Yet Another Resource Negotiator)**: A framework for job scheduling and cluster resource management.

- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

We followed the tutorial on [120] and [121] to set up an Hadoop cluster of 1 namenode (master) and 20 datanodes (slave nodes). The namenode was housed on a stand-alone server with a 4 TB disk, 16 GB RAM, and 8 CPUs. The datanodes are virtual machines that were set up on a testbed on the Virtual Infrastructure for Network Emulation (VINE) [122]. Each datanode consisted of a 100GB disk space, 8GB RAM, and 4 CPUs. Figure 5.1 shows a snapshot of the VINE Web interface for managing the datanodes.
5.1.1 Step 1: Preliminary Checks and Hadoop Installation

First, we made sure to perform preliminary checks and installation such as making sure an acceptable version of Java was installed on the server and disabling IPv6 due to potential address binding errors [121]. Once we completed the preliminary steps, we installed Hadoop (version 3.0.0) and added the appropriate environment variables (e.g. JAVA_HOME and HADOOP_HOME) to the bashrc file.

5.1.2 Step 2: Host File Configuration and Key Generation

We updated the linux hosts file to reflect the IP addresses of each node in our cluster. We used the following convention:

IP-Address-1 prefix-namenode1.cs.fit.edu namenode1 prefix-namenode1
IP-Address-2 prefix-datanode1.cs.fit.edu datanode1 prefix-datanode1

Figure 5.1: Snapshot of the VINE Web Interface
Next, we generated public and private keys for the master node. Since the `hduser` on the master account must be able to SSH (Secure SHell) into its own account as well as the slave nodes, we used the following SSH command to copy the public key from the master to the slave nodes:

```
hduser@namenode1:$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hduser@datanode1
```

After completing this step, we manually SSH’ed into each slave node from the master node in order to add the key fingerprint of each slave to the `known_hosts` file on the master node.

### 5.1.3 Step 3: Hadoop Configuration

This step involved configuring each node in the cluster. First, we configured `hadoop-env.sh` to reflect the appropriate Java home folder. On the master node, we configured a slave and master text-file with the names/aliases of the respective nodes. Further, it is important that we update four main XML configuration files on all machines in the cluster. The names of these files are as follows: `core-site.xml`, `mapred-site.xml`, `hdfs-site.xml`, and `yarn-site.xml`. Specifically, the following properties in each file were updated to reflect critical information such as the address of the jobtracker/namenode, the default Hadoop file system (defaultFS) address, and number of nodes (dfs.replication) on which to replicate files before they become available, and the memory size and number of cores available to each node.

**mapred-site.xml**

- `mapreduce.jobtracker.address`
- `mapreduce.framework.name`
5.1.4 Step 4: System Verification

In this step, we formatted the Hadoop file system using the following command and tested the cluster to make sure it was configured properly.

```
hduser@master:/usr/local/hadoop$ bin/hadoop namenode -format
```
Below is the command we ran to start the cluster as well as a snapshot of the output message we received. In addition, we ran the classic Hadoop WordCount example with *The Project Gutenberg EBook of Moby Dick*¹ as input to make sure our system could execute a job successfully.

```
hduser@master:/home/***$start-dfs.sh
Starting namenodes on [namenode1*****]
namenode1*****: starting namenode, logging to *****namenode1.out
datanode2: starting datanode, logging to *****datanode2.out
datanode4: starting datanode, logging to *****datanode4.out
datanode3: starting datanode, logging to *****datanode3.out
datanode1: starting datanode, logging to *****datanode1.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to *****namenode1.out
```

¹http://www.gutenberg.org/files/2701/2701-0.txt
Chapter 6

Modeling and Classification

This chapter discusses the modeling and classification phase of our work. It provides a detailed explanation on data representation and feature extraction. Included is a discussion on the feature extraction algorithms, the steps followed to prepare training data, and a brief overview of the classifiers evaluated in the work. Portions of this chapter have been published in [123].

Figure 6.1 captures the model building phase of our project.

As can be seen from the diagram, NVD data are fed into our data generator to help us create a set of classifiers for recognizing a set of CWE vulnerabilities. In addition,
the model accepts open source program code, which are analyzed and used for feature
eextraction. In section 6.2, we discuss the features we selected for recognizing each
vulnerability. In Section 6.4, we discuss the classifiers we used to classify the selected
vulnerabilities.

6.1 Data Representation

Each Java file featured in our work is modeled as an Abstract Syntax Tree (AST).
An Abstract Syntax tree is an hierarchical intermediate representation of a program
that presents source code structure according to the grammar of a given programming
language [124]. It is a reduced parse tree where nodes are connected through parent-child
relationships. The main difference between a parse tree and an abstract syntax tree is
that ambiguous grammars result in multiple parse trees for a certain piece of source code
whereas only one AST can be derived from the same piece of code. The construction
of an AST begins with a node that represents the entire translation/compilation unit
followed by a number of intermediate levels, then simple language constructs such as
type name, identifier name, or operator as the leaf nodes [124].

We utilized the JavaParser library to construct and traverse an AST from Java source
code. JavaParser is an open source library that allows native Java interaction with an
AST generated from Java source code [125]. We chose JavaParser due to its ease of
use and simplicity of tree representation. While only one AST may be derived from a
particular piece of code, some tools (e.g. ANTLR) include more detailed information
in their AST representation, resulting in large and complex trees. On the other hand,
JavaParser uses data summarization to obtain simpler trees [126]. JavaParser was built
using the Visitor pattern, which is common in compiler theory. Visitors provide the
ability to navigate an AST [125]. A visitor implements an interpretation (an object
which contains a visit method for each syntax tree class) [127]. Each syntax-tree
class contains an accept method that serves as a hook for all interpretations [127].
Listing 6.1 shows a snippet of the JavaParser implementation of the visitor pattern in the `MethodCallExpr` class.

Listing 6.1: Example Implementation of the Visitor Pattern in JavaParser

```java
public final class MethodCallExpr extends Expression implements
    NodeWithTypeArguments<MethodCallExpr>,
    NodeWithArguments<MethodCallExpr>,
    NodeWithSimpleName<MethodCallExpr>,
    NodeWithOptionalScope<MethodCallExpr> {

    ...

    public <R, A> R accept(GenericVisitor<R, A> v, A arg) {
        return v.visit(this, arg);
    }

    public <A> void accept(VoidVisitor<A> v, A arg) {
        v.visit(this, arg);
    }

    ...
}
```

ANTLR (ANother Tool for Language Recognition) is an open-source parser generator that reads, processes, executes, or translates structured text or binary files [128]. In addition, ANTLR can be used to generate a parser from a given language grammar and can be used to build and walk parse trees and ASTs [128].

We will now use an example code snippet to show the difference between ASTs produced by ANTLR and JavaParser. In addition to its Java object representation, ANTLR also outputs a graphical AST of the code being modeled. In contrast, JavaParser does not support graphical tree representation. Using a set of visitors, we were able to
generate a tabbed, textual version of the JavaParser AST, which we then used to generate a graphical tree.

Listing 6.2: Example Java Code

```java
import java.sql.*;

class Login {
    
    public boolean doLogin(String username, String pwd) {

        String sqlString = "SELECT * FROM db_user WHERE username = \"" + username + \"\" AND password = \"" + pwd + \"\";
    }
}
```

From code Listing 6.2, ANTLR produced the AST shown in Figure 6.2. Figure 6.3 shows a graphical representation of the AST generated using JavaParser based on the same code listing. As can be seen from the trees, JavaParser produces a simpler, more navigable tree.

### 6.2 Feature Extraction

In our work, we focus on the top 2 CWE. In this section, we discuss the feature sets we extracted in order to recognize and classify each vulnerability. A feature set is a collection of the outputs generated by the feature extraction algorithms[129].
Figure 6.2: ANTLR AST Example
Figure 6.3: JavaParser AST Example
6.2.1 MapReduce Algorithm For Feature Extraction

The MapReduce algorithm that we implemented for execution in Apache Hadoop is shown in Algorithm 1. Considering that the projects in Sourcerer 2011 have an inconsistent organization structure, each project must be processed to ensure that files follow a uniform structure that can be submitted to the Hadoop File System for processing. First, we wrote a Bash script to parse each project.properties within each project in the repository, and utilized Bash utilities to extract information about each project in order to create a more uniform file structure. Java files were reorganized such that there is one directory for each project. The filenames were later used as keys for the MapReduce framework. To create a unique filename, each repo URL and a universally unique identifier (UUID) were prepended to the original file name of the Java files within each project directory. For example, given a repo project identified by the url: google-api-translate-java.googlecode.com and a Java file with the name DBHelper.java, the fileID (key) would resemble the following: google-api-translate-java.googlecode.com-74b000bc-06ba-11e8-83ee-000c29bc3179~DBHelper.java. An added benefit of this convention is to provide support for validating automatically-labeled data and querying the dataset during the modeling and application phases of our project.

Further, Hadoop utilizes blocks and input splits for data processing. Blocks refer to physical storage locations and the default block size is 64MB [130]. The default split size is equal to a block size. Users can write custom input split functions to specify how files should be handled in HDFS. Since each Java file is modeled as an AST, it is important that HDFS process each file without splitting it. To ensure this, we wrote a custom record reader that reads each Java file and submits it to the mappers for processing as shown in the algorithm. Each file is then transferred from the master node to the slave node determined by the YARN resource manager for processing.
Algorithm 1: MapReduce Algorithm for Mining Features from Java Code

**input**: repository.path: path to repository dataset
**output**: a set of features for a certain vulnerability

1. **foreach** project ∈ repository.path **do**
   2. javaDataFiles = selectJavaFiles()
   3. createCustomRecordReader() // record reader to read full java program file

4. **foreach** javaFile ∈ javaDataFiles **do**
   5. **Function** map(javaFile):
      6. key = getFileName(javaFile)
      7. value = extractText(javaFile) // using customRecordReader
      8. addToIntermediateList(key, value)
      9. emit(intermediateList)

   /* each reduce is a vulnerability detector that emits a set of features for identifying a certain vulnerability */

10. **Function** reduce(intermediateList):
    11. **foreach** pair ∈ intermediateList **do**
        12. outkey = intermediateList.key
        13. inValue = intermediateList.value
        14. outValue = buildFeatureSet(inValue) // based on abstract syntax tree
        15. emitFinal(outKey, outValue)
6.2.2 Extracting Features for Classifying CWE/CVE Vulnerabilities

We now provide a brief discussion on the top 2 CWE that we study in this work as well as the features we use for detecting and classifying them. More detailed information on these vulnerabilities can be found on the CWE website [3]. The vulnerabilities are ordered using the CWE ranking presented in Table 4.1. The two vulnerabilities that we investigate are CWE-89 and CWE-78, which fall into the category of taint-style or information flow vulnerabilities [131, 132]. These are vulnerabilities that are caused by the lack of input/output validation and are traditionally modeled as source-sink problems.

The concept of source and sink has been discussed widely in the literature [131, 133, 134, 135]. A source, in the context of information-flow vulnerabilities, refers to an untrusted data source from which user input is received [135]. A sink is a security-sensitive function (e.g., the java.sql.Statement.executeQuery function) [135]. A source-sink error arises if a value constructed at a location designated as a source reaches a location designated as a sink [133]. Using online resources [136, 137, 138], we were able to collect documentation that discusses sources and sinks in Java code. From these articles, we compiled a set of possible values for our source and sink features. Some static analysis tools perform dataflow analysis by either requiring users to specify the sources and sinks or by automatically detecting them using techniques such as byte-code analysis [133]. It would be interesting to ascertain whether the presence/absence of these known methods could impact the performance of the classifiers we apply to detect the vulnerabilities.

To create a corpus for training and evaluating classification algorithms, we first parsed Java code and extracted all methods and statements. Next, we checked for statements related to a particular vulnerability in the set of all statements and captured their respective methods. Features relating to a particular vulnerability are extracted from
the captured information and returned as Comma Separated Values (CSV). Metadata is also included as a feature in each feature set. In our case, metadata is a base-64 encoding of statements and methods that were used in creating the feature set. The data is encoded in order to escape CSV delimiters that are found in the source code. First, we formatted the data using XML, so that it can be easily parsed during model verification. Further, for information-flow vulnerabilities, we checked for the presence of tainted sources and sinks in the method call expressions. A FeatureSetBuilder Java class is used to aggregate the features and output them as CSV.

6.2.2.1 CWE-89 – Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)

An SQL Injection (SQLI) attack occurs when an attacker provides specially crafted input to an application that employs database services such that the provided input results in a different database request than was intended by the application programmer [139]. SQLI has been a common vulnerability for many years, securing position number one on the OWASP 2010 [140], 2017 [141] and the CWE 2011 [3] lists. Applications (e.g. web-apps) generally accept user input, which are then used in executing database requests. These requests are typically SQL statements. SQLI is a serious vulnerability because it could lead to unauthorized access to sensitive data, cause severe updates to or deletions from a database, and even result in devastating shell command execution [131].

The example code featured in Listing 6.2 is an example of code that can potentially lead to an SQL Injection. This is due to the fact that the doLogin function accepts the parameters username and password in plaintext and includes them in the SQL query string without neutralization. The use of the PreparedStatement class from JDBC or J2EE is often recommended as a fix for SQL Injection [142]. This class allows for the use of a wildcard (“?” character) to create a parametric query that escapes potentially tainted user input. However, improper use of this class could still lead to SQL Injection. In the snippet of code in Listing 6.3, the PreparedStatement class is used, but the query
could still be tainted because of unsanitized input. Listing 6.4 shows a compliant query that mitigates against SQL Injection.

Listing 6.3: Example Java Code that could potentially lead to SQL Injection

```java
import java.sql.PreparedStatement;
class Login {
    public boolean doLogin(String username, String pwd) {

        String sqlString = "select * from db_user where username=" + username + " and password =" + pwd;
        PreparedStatement pStmt = connection.prepareStatement(sqlString);
        ResultSet rs = pStmt.executeQuery();
    }
}
```

Listing 6.4: Example Java Code that mitigates SQL Injection

```java
import java.sql.PreparedStatement;
class Login {
    public boolean doLogin(String username, String pwd) {

        String sqlString = "select * from db_user where username=? and password=?";
        PreparedStatement pStmt = connection.prepareStatement(sqlString);
        pStmt.setString(1, username);
        pStmt.setString(2, pwd);
        ResultSet rs = pStmt.executeQuery();
    }
}
```

If we compare Listing 6.2 with Listing 6.4, a number of differences can be noted between the two samples of code that can be used to create a feature set for recognizing
the SQL injection vulnerability. First, the unsafe query string in Listing 6.2 contains explicit apostrophes, and no methods are used to sanitize the user input. Consequently, we identify six main features for detecting and classifying the vulnerability, which are presented in Table 6.1. Two of these features are multivalued attributes (source and sink) and, therefore, can be simplified into boolean attributes during evaluation of certain classifiers.

Since the main cause of SQLI is unsanitized input and un-escaped apostrophes, the features quoted variables found and potentially sanitized can be very effective in the detection of the vulnerability. However, simply checking the presence of an apostrophe is not acceptable because a query string such as `iper.update("insert into people values('namtesss','sdfjlsfjls')", conn)`, which was found in a unit test of the `goldriver.googlecode.com` open source project in the Sourcerer 2011 dataset, is a safe SQL statement. It is the quoted portion of a concatenated query string that should not contain single quotes as discussed before and depicted in Listing 6.2. In addition, prepared statement imported and all queries parameterized can help us to measure the utility of the PreparedStatement class in mitigating SQLI across the projects in the Sourcerer 2011 dataset.

The BuildFeatureSet procedure for creating a feature set for SQLI is shown in Algorithm 2. Using the list of generated features, a data instance in the ground truth corpus is automatically labeled as safe if the boolean feature quoted variables found is false, the incoming variables are potentially sanitized, and parameterized queries are used to create the SQL statements. A variable is considered potentially sanitized if it is passed to a function that is not in the list of known tainted functions. If these properties are not satisfied, the data instance is labeled as unsafe.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Data Type</th>
<th>Possible Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sources</td>
<td>multi-valued</td>
<td>{getPathInfo, getResource, getName, getServletPath, getRemoteHost,</td>
<td>The method that accepts or processes potentially tainted user input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getLocalAddr, getParameterMap, getRealPath, getServerName, getPathTranslated,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getInitParameterNames, getHeader, getCookies, getPath, getComment, getParameter,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getParameterValues, getRequestURL, getHeaders, getRequestURL, getResourceAsStream,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getRequestDispatcher, getQueryString, getResourcePaths, getDomain, getValue,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getLocalName, getInitParameter, getRemoteUser, getHeaderNames,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getContentType, getParameterNames, concatenWhere, getNamedDispatcher}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>executeLargeUpdate, updateWithOn-Conflict, setGrouping, queryForList,</td>
<td>The method that creates, modifies, or executes a SQL query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>batchUpdate, update, buildQuery, prepareStatement, delete, buildUnionSub-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Query, queryWithFactory, rawQueryWithFactory, nativeSQL, queryForInt, blobFileDescriptorForQuery, longFor-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Query, sqlRestriction, newQuery, executeInsert, createQuery, queryForMap,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>queryForLong, apply, execSQL, query-ForRowSet, query, stringForQuery,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>buildQueryString, &lt;init&gt;, addBatch, execute, executeQuery, createSQLQuery,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>createNativeQuery, setFilter, appendWhere, queryForObject, new-PreparedStatementCreator, as, compileStatement, createDbFromSqlStatements, buildUnionQuery, rawQuery,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>executeUpdate, prepareCall}</td>
<td></td>
</tr>
<tr>
<td>quoted_variables_found</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Tells whether explicit apostrophes were used to formulate an SQL query string</td>
</tr>
<tr>
<td>potentially_sanitized</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Tells whether user inputs were passed to untainted functions before being used in SQL strings</td>
</tr>
<tr>
<td>prepared_statement_imported</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Specifies whether the recommended PreparedStatement class was imported</td>
</tr>
<tr>
<td>all_queries_parameterized</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Specifies whether the question-mark wildcard was used as variable placeholders in query strings</td>
</tr>
<tr>
<td>METADATA</td>
<td>String</td>
<td>–</td>
<td>Data (encoded in base 64) containing SQL statements and methods found in each Java file to assist with verification of classification</td>
</tr>
<tr>
<td>class</td>
<td>binary</td>
<td>{SAFE, UNSAFE}</td>
<td>The target variable</td>
</tr>
</tbody>
</table>
Algorithm 2: Procedure for Building the Feature Set for Detecting SQL Injection

**input**: ast: abstract syntax tree of Java code  
**output**: a set of features for detecting SQL Injection

1. **Procedure** `buildFeatureSet(ast)`

2. Initialize featureSet parameters as safe

3. `imports = Get list of ImportDeclaration from ast`

4. `sqlStatements = Extract all statements containing SQL Commands from ast`

5. `methodCalls = Get list of MethodCallExpr from ast`

6. `sources = Get list of all tainted sources from imports`

7. `sinks = Get list of all sinks from methodCalls`

8. Set feature `sources = sources`

9. Set feature `sinks = sinks`

10. `foreach sqlStatement ∈ sqlStatements do`

11.  `if sqlStatement is concatenated string then`

12.   `Create stmtArray from sqlStatement`

13.   `foreach item ∈ stmtArray do`

14.    `if item is functionCallExpr & item ∈ taints then`

15.     `Set feature potentially_sanitized = false`

16.    `if item is variable & item not passed to potential sanitizer function then`

17.     `Set feature potentially_sanitized = false`

18.    `if item is string & item contains apostrophes then`

19.     `Set feature quoted_variables_found = true`

20.  `if all sqlStatements parameterized then`

21.   `Set feature all_queries_parameterized = true`

22.  `if preparedStatement class found ∈ imports then`

23.   `Set feature prepared_statement_imported = true`
6.2.2.2 CWE-78 – Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)

Command injection is an attack in which the goal of the attacker is to execute arbitrary commands on the host operating system via a vulnerable application [143]. As the name suggests, these commands are typically targeted to the command shell, which is a software program that provides direct communication between the user and the operating system [144]. The commands supplied by the attacker are usually executed with the same privileges of the vulnerable application.

As a type of taint-style vulnerability, command injection is more targeted to web applications than other software due to the multiplicity of attack vectors. For example, commands may be injected into form-data, cookies, and HTTP headers. However, general-purpose applications are not exempted. All Java applications have an instance of the Runtime class that allows the application to interface with the environment in which it is running [145]. Programmers can use the Runtime.exec(...) method to execute commands. The C programming language provides the (system(command)) method that allows programmers to pass arguments to the shell (e.g.: `/bin/sh`) to be parsed and executed. C code is more prone to command injection than Java because Runtime.exec does not try to invoke the shell at any point unless it is explicitly supplied as the command by the programmer [143]. Instead, Runtime.exec tries to tokenize the input string into an array of strings and operate on the array accordingly [145]. Essentially, the array is treated as follows: cmdarray[0] contains the command to be executed and cmdarray[1]···cmdarray[n − 1] contain parameters to the command.

Listing 6.5 shows an example of benign code that uses Runtime.exec to invoke the Windows find command with user arguments. Because this command is an executable program, even if the user-supplied arguments contain executable code, Runtime.exec will pass all arguments as parameters to the find utility. However, the code in Listing 6.6 is prone to command injection. This is because it utilizes the Windows command
shell (`cmd.exe`) to execute the `dir` command without proper sanitization. For instance, if the user inputs `" . & echo hi!"`, this would be a form of command-chaining, which would cause the command shell to execute the `echo` command in addition to the `dir` command.

Listing 6.5: Example of Safe Java Code that Uses Runtime Exec

```java
import java.io.*;
class FindIt {
    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        Process proc = runtime.exec("find" + " " + args[1]);
    }
}
```

Listing 6.6: Example of Unsafe Java Code that Uses Runtime Exec

```java
import java.io.*;
class ChangeDir {
    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        String[] cmd = new String[3];
        cmd[0] = "cmd.exe";
        cmd[1] = "/C";
        cmd[2] = "dir " + args[0];
        Process proc = runtime.exec(cmd);
    }
}
```
Another example found on [146] is as follows:

$ java Exec 'sh -c $@|sh . echo /bin/echo -e "tab\trequired"'

In essence, on Linux systems, the special parameter, $@, may be used to manipulate the parameters supplied to the shell to execute chained commands. Coupled with encodings of whitespace, such as \t, a sequence that seems harmless could result in command injection. The following are some recommendations that have been made to prevent or mitigate command injection:

- Use `java.io.FilePermission` in the Java SecurityManager to restrict file permissions [147]
- Restrict directories to a certain type [148]. For example:
  
  ```java
  Integer.parseInt(System.getProperty("dir"))
  ```
  restricts the user to a set of numeric directories when executing the `dir` command
- Sanitize the user arguments [149]
- Use an API that offers proven security against command injection [143]. For example, the Java `File` class allows users to list files (`java.io.File.list()`) within a directory instead of executing the OS `ls` or `dir` command.

By using these recommendations, we present four main features for detecting command injection in Java programs. These features are described in Table 6.2. In addition, we propose the procedure in Algorithm 3 for building the command injection feature set.
Table 6.2: Features for Recognizing OS Command Injection

<table>
<thead>
<tr>
<th>Feature</th>
<th>Data Type</th>
<th>Possible Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shell_command_presents</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Tells whether a shell command is supplied to runtime.exec. Shell commands include command.com, cmd.exe, /bin/sh, /bin/csh, /bin/ksh, /bin/bash, /bin/tcsh, /bin/zsh, /bin/rc, /bin/es.</td>
</tr>
<tr>
<td>unsanitized_args_presens</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Specifies whether the programmer passes potentially tainted user arguments to the runtime.exec method.</td>
</tr>
<tr>
<td>faulty_characters_presents</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Specifies whether faulty characters are present in the command passed to the runtime.exec method.</td>
</tr>
<tr>
<td>file_permission_imported</td>
<td>boolean</td>
<td>{TRUE, FALSE}</td>
<td>Tells whether the recommended Java File permission class is imported to prevent command injection.</td>
</tr>
<tr>
<td>METADATA</td>
<td>String</td>
<td>–</td>
<td>A field containing runtime examples and methods found in each Java file.</td>
</tr>
<tr>
<td>class</td>
<td>binary</td>
<td>{SAFE, UNSAFE}</td>
<td>The target variable.</td>
</tr>
</tbody>
</table>

From the feature set, the following heuristic can be used to automatically label the ground truth corpus for command injection: if shell commands are present and unsanitized arguments/variables are used in the command string or any of the faulty characters mentioned before are used in the command string, label the data instance as unsafe. Otherwise, label the instance as safe.

6.3 Preparing Training Data

To prepare training data for classifying each vulnerability, we implemented our MapReduce algorithm in Java and executed it in Apache Hadoop. The data in Part a of the Sourcerer 2011 dataset was used to create the corpus for evaluating the classification algorithms. Table 6.3 summarizes the distribution of the projects within the subset of the dataset that was analyzed. Specifically, the Sourceforge projects and Google Code projects were processed to create training data and test data, respectively. Table 6.4 shows the breakdown of the training and testing samples.
Algorithm 3: Procedure for Building the Feature Set for Detecting Command Injection

input: ast: abstract syntax tree of Java code

output: a set of features for detecting Command Injection

Procedure buildFeatureSet(ast)

1. Find (all ExpressionStatements ∈ the exec method) ∈ ast

2. foreach cmd parameter ∈ exec statement do

3. // the cmd parameter is the 1st parameter based on the exec method signature

4. if cmd parameter is concatenated string then

5. updateFeatureSet(cmd)

6. else

7. Find all occurrences of cmd variable in ast

8. if any occurrence is string then

9. updateFeatureSet(cmdOccurrence)

Function updateFeatureSet(cmdString):

10. if concatenated variables ∈ cmdString ∼ potentially sanitized then

11. Set feature unsanitized_args_processed = true

12. if cmdString ∋ a call to a shell command then

13. Set feature shell_command_present = true

Table 6.3: Distribution of projects in part “A” of the Sourcerer dataset

<table>
<thead>
<tr>
<th>Repository</th>
<th>Number of Projects</th>
<th>Number of Java Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Code</td>
<td>6,865</td>
<td>605,809</td>
</tr>
<tr>
<td>Sourceforge</td>
<td>7,511</td>
<td>1,015,732</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>-</td>
<td>625,302</td>
</tr>
<tr>
<td>Total</td>
<td>14,376</td>
<td>2,246,843</td>
</tr>
</tbody>
</table>

More details on our environment set-up can be found in Chapter 5. The output from Hadoop is a CSV file which we converted to the ARFF file format for processing in Weka. In addition to the command-line utility, Apache provides a web UI for us to track each MapReduce job as well as to view the output logs for troubleshooting purposes. Figure 6.4 shows a snapshot of a job status being displayed in Apache Hadoop.
Table 6.4: Breakdown of training and testing data

<table>
<thead>
<tr>
<th>Training &amp; Testing Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQLI Corpus</strong></td>
</tr>
<tr>
<td>Repository</td>
</tr>
<tr>
<td>Sourceforge</td>
</tr>
<tr>
<td>Google Code</td>
</tr>
<tr>
<td><strong>Command Injection Corpus</strong></td>
</tr>
<tr>
<td>Sourceforge</td>
</tr>
<tr>
<td>Google Code</td>
</tr>
</tbody>
</table>

6.4 Classifiers

Three classifiers were evaluated in this work in order to select one for adoption into the proposed methodology. In the next sections, we briefly describe each classifier.

6.4.1 Decision Trees

Decision trees are widely used in machine learning and operations research. In machine learning, they are used in supervised learning to predict the outcome of a test based on a set of training examples. The goal is to partition a dataset into homogeneous groups using concepts from information theory based on a target variable or prediction. Descriptively, a decision tree is a model that resembles a flowchart/orientation diagram in which each internal node represents a test on a feature or attribute, each branch represents the
outcome of the test, and each leaf node represents a class or target label\[150\]. There are several decision tree algorithms, such as ID3 (Iterative Dicotomizer 3)\[151\], C4.5 \[152\], and C5.0 (a commercial version of C4.5). In this work we used the Weka\[153\] implementation of C4.5, which is known as J48.

6.4.2 Random Forests

A random forest is a classifier that consists of a collection of decision trees where each tree casts a unit vote for the most popular class for a given input \[154\]. As a decision tree ensemble method, each tree is learned using bootstrap aggregating (bagging), where a sampling of the original training data is done with replacement and a subset of features are chosen for each tree.

It has been shown in the literature \[155, 156\] that ensemble methods perform better than decision trees on many datasets. This is due to that fact that decision trees may be overly large and tend to overfit the training data and thus perform poorly on unknown instances. Unlike decision trees that have to be pruned to achieve their maximum performance, the trees in a random forest are not pruned. Instead, all trees participate in weighted voting to determine the classification of a given data instance.

6.4.3 Support Vector Machines

Based on principles from statistical learning theory, Support Vector Machines (SVMs) are a set of related methods that are used in supervised learning to solve classification and regression problems \[157\]. SVMs work by creating a maximum-margin hyperplane that splits examples by their target variables while maximizing the distance to the nearest cleanly split examples. The most common type of SVMs is linear in nature, and attempts to divide classes of examples in feature space using a linear model. More generally, non-separable SVM models that use kernel functions (e.g. a radial basis function) have been proposed to classify noisy samples using a nonlinear decision boundary.
Chapter 7

System Design and Implementation

This chapter describes the design and implementation of the system. First, we present our initial ideas on the requirements and design of a useful and effective code analysis tool. Next, we summarize the results of a knowledge elicitation survey that was conducted to empirically ascertain the current use of code analyzers among programmers and to elicit their views on the design of our system. We then discuss the impact of the survey on the final design of the system.

7.1 Initial System Design

After reviewing the literature and a selection of popular code analyzers such as FindBugs and CheckStyle, we came up with a list of ideas that should be considered when designing a code analyzer. As discussed in Chapter 2, the lack of acceptance and poor use of code analyzers have led to a great deal of effort being invested in the improvement of alerts/recommendations provided by existing analyzers. We also determined that if the
code analyzer is separate from the development framework (IDE), it is unlikely that programmers will use it. Consequently, we came up with the following inexhaustive set of requirements:

- The system must be a part of the IDE to enable effective scanning and mediation
- The warnings should be brief and actionable (Links to more detailed information should be provided for interested users)
- Emphasis should be placed on fixing the potential vulnerabilities and encouraging good programming practice
- The fixes should not be generic but as specific as possible to the project being developed
- Scanning of vulnerabilities should be done such that the programmer’s productivity is not negatively impacted

By taking these requirements into consideration, we came up with the mockup shown in Figure 7.1. The proposed tool is called VulIntel, short for IntelliSensing Vulnerabilities. The tool is intended to be part of the IDE and should use IntelliSense technology to scan code as the programmer types. A list should be populated with the names/IDs of potential vulnerabilities. Clicking on a vulnerability in the list should display a brief description of the vulnerability including a reference to the unsafe method and variables involved. Further, a ranked list of examples should be presented to the user to help with mitigation.

After creating the initial design, we conducted a knowledge elicitation survey to collect feedback from programmers on the design of the system and also to understand how and to what extent programmers are using current code analyzers. We summarize the results from the survey in the next section.
7.2 Knowledge Elicitation Survey

This section presents empirical results from a knowledge elicitation survey (See Appendix A for survey questions) that was conducted to obtain formative feedback on the design of an interface and the views of programmers about a tool that utilizes IntelliSense technology to find vulnerabilities in program code and provides recommended fixes for detected vulnerabilities. It is important to solicit feedback for any system design to satisfy usability requirements as well as to answer questions that will assist with development. Since the survey involved human subjects, approval\(^1\) was first obtained from the Institutional Review Board (IRB) at Florida Institute of Technology to ensure

\(^1\)IRB#: 18-006
minimal risk to participants. The survey sought to answer questions that included but were not limited to the following:

1. To what extent are programmers using code analyzers?

2. How useful are the advice/recommendations provided by existing tools?

3. Would programmers utilize a tool that uses IntelliSense technology to find and suggest fixes for vulnerabilities?

4. What are the design criteria and expectations for a tool that scans code for vulnerabilities and presents fixes to the user?

7.2.1 Participants

To recruit a diverse population of participants, emails were sent to individuals of various ranks in industry and academia. The list consisted of more than 10 organizations that included the Palm Bay (Florida) division of Harris Corporation, CodeCraftWorks, Intel, Seattle Alexa division of Amazon, Florida Institute of Technology, universities in New York, Baltimore, Brazil, Germany, and the UK, among many others. The main criteria for participants was that they have at least 3 years experience with an object-oriented programming language such as Java, C#, or C++. The individuals contacted for the survey were told to share the link to the survey with individuals meeting the criteria at the various institutions. Participants were told that the duration for the survey would be approximately 20 minutes. Further, they had to agree to an informed consent form before completing the survey. A total of 104 participants completed the survey (44 graduate students, 39 industry professionals, 11 undergraduate students, 7 professors, and 3 others).
7.2.2 Familiarity with Programming Languages and IDEs

Participants were asked to select their familiarity with a set of programming languages from a list that uses a five-point Likert scale\(^2\). Of the languages, participants were most familiar with the Java Programming language, with 40% indicating that they are “very familiar” with it and 25% claiming to be “experts” (See Figure 7.3). The IDE that scored the highest in popularity (84.62%) among participants was Eclipse. This was followed by Visual Studio with 73.08% and Netbeans with 61.52%. The results are summarized in Figure 7.2.

Figure 7.2: Participants’ familiarity with IDEs

\(^2\)“None”, “Somewhat familiar”, “Familiar”, “Very familiar”, “Expert”
Figure 7.3: Participants' familiarity with programming languages
7.2.3 Use of Existing Code Analyzers

To answer question 1 and 2 mentioned above, participants were asked whether they performed static and/or dynamic analysis on their code and how useful they found the given recommendations. 13.46% of the participants stated that they used a static analyzer such as FindBugs, 3.85% used a dynamic analyzer such as Java PathFinder, 9.62% used both dynamic and static analyzers, and 56.73% reported that they did not scan their code for vulnerabilities. Of the respondents who used a code analyzer, 25.81% described as “helpful” the recommendations they received from the scanners they used and 67.74% reported that the advice given was “somewhat helpful” in fixing vulnerabilities.

7.2.4 Views and Expectations Regarding the Proposed tool that IntelliSenses Vulnerabilities

To introduce participants to the proposed tool and to elicit knowledge regarding their use of IntelliSense technology and opinions about the proposed design, they were first asked if they currently take advantage of IntelliSense technology. 68 participants reported that they currently utilize the technology while 32 did not; 4 participants skipped the question. In addition, the participants were asked their opinion about the application of IntelliSense technology to vulnerability detection. 87 of the participants intimated that they would appreciate a system that can scan their code for vulnerabilities as they code; 10 were not interested in the technology, but believe other programmers may be interested; 3 participants did not believe it would be a good idea to apply IntelliSense to vulnerability detection, and 4 skipped the question. The participants were then shown the mockup (See Figure 7.1) of the proposed tool and asked in what situations and for what types of projects they would utilize it. The responses are summarized in Figure 7.4 and 7.5. Moreover, they were asked their opinion about what they (dis)liked about the interface and what types of vulnerabilities they would like to detect using the tool.

78
Figure 7.4: Situations under which programmers would use the proposed plugin

Figure 7.5: Types of projects for which programmers would use the proposed plugin

7.2.5 Themes that Emerged from the Survey

Several important themes stood out in the responses provided by participants in the knowledge elicitation survey. From the list of vulnerabilities provided by the participants, SQL injection, buffer overflows, and the OWASP list of vulnerabilities are well-known
and important to programmers. However, there are other vulnerabilities that are often overlooked by programmers but could pose significant risks. For example, Figure 4.3 shows that hard-coded credentials (CWE-789) and missing encryption (CWE-311) account for dozens of vulnerabilities in the 2017 NVD release, yet these vulnerabilities were not mentioned by any participant.

While a significant percent of the participants (56.73%) reported that they do not currently take advantage of code analyzers, 87% stated that they would appreciate a tool that can scan code for vulnerabilities as they write code. Further, a significant 61% (See Figure 7.4) indicated that they would use the proposed tool to scan for vulnerabilities while coding.

Three main themes emerged from the open-ended responses that were provided by the participants:

**Usability:** Some participants were concerned about the number of objects on the proposed UI. They suggested that while updates are important, the “news updates” panel adds clutter to the interface and should be minimized if possible.

**Performance:** While some participants were in favor of scanning being done in the background, a few of them were concerned about the impact this may have on the code editor. For example, one participant submitted the following response:

“I like that it tells you security vulnerabilities as you type. I am a little concerned about how efficient scanning for these vulnerabilities might be. I would most likely stop using it if it slowed down my editor.”

**Fixing Vulnerabilities:** A number of participants commented on the plugin’s proposed ability to provide fixes for the vulnerabilities that it finds. One participant provided the following feedback:
“Really helpful as it provides you with multiple fixes and examples and visually appealing.”

7.3 System Architecture

Figure 7.6 shows a detailed description of the system architecture. It captures the relationship between the model building phase and the application phase. It represents a more detailed model of the diagram presented in Chapter 3. As shown in the diagram, the code analyzer, which is part of the modeling phase, is connected to the application phase via the recommender system that accepts the code from the user interface and uses a classifier to make recommendations to the user.

![Diagram of system architecture]

Figure 7.6: Final system framework highlighting the application phase

7.4 Final System Design

First, a classification model was created using the Weka API by incorporating the knowledge extracted from open source projects as discussed in Section 6.2. The model
was serialized and imported into an Eclipse plugin. Figure 7.7 shows a screenshot of the final system as an Eclipse plugin. The design of the plugin was influenced by the responses received in the knowledge elicitation survey. For example, the “updates” panel (See Figure 7.1) was removed from the user interface. Intellisense technology was utilized by extending the Eclipse Code Recommenders\(^3\) system, which is a fundamental component within the Eclipse intelligent code completion framework. The Intellisense system was programmed to initiate the scanner after the user enters or removes at least 5 characters, excluding spaces. This behavior was chosen after experimenting with options such as after method completion or after entering or removing at least 10 characters.

\(^3\)http://www.eclipse.org/recommenders/

Figure 7.7: Screenshot of the Final Design of the plugin as incorporated in the Eclipse environment
7.5 Recommending Fixes

The ultimate goal of detecting vulnerabilities is for programmers to fix them and produce more secure code. Since the proposed knowledge extraction algorithms create labeled corpora for detecting a set of vulnerabilities, it is of interest to use the vulnerability-safe (negative) examples to provide recommendations to help programmers fix the detected vulnerabilities. Several questions arise in determining a similarity scheme that finds code that is similar to the user’s code but is safe against the vulnerabilities found in the user’s code. For example, what is the best trade-off between the time taken to find similar code that is not only syntactically relevant but also semantically helpful to the user? To answer this and other questions, experiments were conducted using three text similarity schemes (cosine similarity, MinHash, and SimHash) in order to select one that takes the least amount of time to find relevant examples.

Below we briefly discuss the three similarity schemes and present the results of an experiment that was conducted to determine the most appropriate metric to employ to provide vulnerability mitigation examples to the user. While the number of fixes for a certain vulnerability may be small, it is important to select a similarity scheme that is fast enough to be adaptable to the IntelliSense methodology.

7.5.1 Cosine Similarity

The cosine similarity between two vectors (or two programs) is a measure that calculates the cosine of the angle between them irrespective of the magnitude of the vectors. It is calculated as the dot product of two numeric vectors, and is normalized by the product of the length of the vectors. Thus, cosine similarity values close to 1 indicate high similarity while those close to 0 indicate low similarity. In this work, the vectors represent the term frequencies of terms that are common between two programs (methods). The
vectors were created by using Apache Lucene\(^4\) to tokenize the Java code and remove Java keywords and other English stop words from the code.

The formula for cosine similarity is given in equation 7.1

\[
\cos(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|}
\]  

(7.1)

### 7.5.2 MinHash

Minhash is a Locality Sensitive Hashing (LSH) technique that is based on the min-wise independent permutations of sets. The goal of MinHash is to estimate the Jaccard similarity quickly without explicitly computing the intersection and union of the sets. Jaccard is the ratio of the number of elements in the intersection of two sets to the number of elements in the union. The idea behind MinHash is to devise a signature scheme such that the probability that there is a match between the signatures of two sets, \(S_1\) and \(S_2\), is equal to the Jaccard measure [158]. The simplest (and relatively fast) version of the MinHash scheme uses \(k\) different hash functions to generate the signatures of the sets. \(k\) is a fixed integer that represents each set \(S\) by the \(k\) values of \(h_{\min}(S)\) for the selected functions [158]. In this work, a \(k\) value of 7 and hash bit length of 32 bits were chosen for the experiment. These parameters were chosen to be consistent with the literature [158].

### 7.5.3 SimHash

SimHash is also a LSH for the cosine similarity measure that maps high-dimensional vectors to small fingerprints [159]. It is based on the concept of Signed Random Projections (SRP) that transforms a multi-dimensional vector into a binary string and stores only the sign of the random projection values.

\(^4\)https://lucene.apache.org/
Figure 7.8: Finding safe code that is most similar to the user’s code

Figure 7.8 presents the results from an experiment that compares the three similarity approaches. First, the figure shows a sample user code that is vulnerable to SQLI. Next, the most similar code that fixes the vulnerability, as returned by each algorithm, is presented. The figure also shows the similarity score and the time taken to search a dataset of 18,842 safe instances for code that is similar to the user’s code. As can be seen from the results, all three algorithms finished the search in under 2 seconds. Moreover, the returned samples suggest that cosine similarity produced a more semantically similar piece of code to the user’s code.
Chapter 8

Evaluation

This chapter provides a tri-fold evaluation of the methodology proposed in this work. First, we evaluate the performance of three classifiers in classifying two taint-style vulnerabilities. Second, we perform a scalability analysis of the methodology. Third, we discuss the results from a usability study in which participants from industry and academia were invited to perform A/B testing of the tool.

8.1 Classifier Performance

Tables 8.1 shows the results of three classifiers (decision tree, SVM and random forest) on detecting SQL injection while Table 8.2 presents the results on detecting command injection. While the corpus for command injection is relatively small and unbalanced, the overall results confirm the premise that classifiers can be trained using features extracted from source code and used to classify code as safe or unsafe. Based on this collective performance, any of the three classifiers could be employed in the final design of the proposed system. Consequently, decision tree (Weka J48) was chosen and used to build the models discussed in Section 7.4.
Table 8.1: Classifier performance for SQLI

<table>
<thead>
<tr>
<th>Class</th>
<th>TPR</th>
<th>FPR</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decision Tree (J48)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td><strong>SVM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td><strong>Random Forest</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.000</td>
<td>0.999</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 8.2: Classifier performance for command injection

<table>
<thead>
<tr>
<th>Class</th>
<th>TPR</th>
<th>FPR</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decision Tree (J48)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>0.996</td>
<td>0.000</td>
<td>1.000</td>
<td>0.996</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.004</td>
<td>0.905</td>
<td>1.000</td>
</tr>
<tr>
<td><strong>SVM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td><strong>Random Forest</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>unsafe</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>
8.2 Scalability

While the goal of this work is to couple machine learning techniques with IntelliSense technology to create a recommender system that detects and mitigates vulnerabilities in user programs, it is also of interest to determine the scalability of the proposed methodology on projects of various sizes. To do so, a random sample of 10 Google Code projects in the dataset was selected and processed for SQL injection. Table 8.3 and Figure 8.1 show the time taken to classify these projects for SQLI by using a Macbook Pro laptop (16GB of RAM, 3 GHz Intel Corei7 processor). The experiment was done while other processes were running on the machine in order to mimic the environment of a typical developer/programmer. The table also provides other information on the experiment such as the total LLOC (logical lines of code) for each project and the number of files in each one. LLOC was computed by counting the expression statements (an expression followed by a semicolon) in each AST.

The results show that the approach scales very well by being able to scan a project of over 4.5 million lines of code for SQLI in under 8 seconds while projects of up to 1 million lines of code take under a second. Even though the experiment was only done for one vulnerability, the scanning process can be parallelized through the use of threads to maintain this performance while scanning for other vulnerabilities.
Figure 8.1: Classification Time vs Number of Java Files

Table 8.3: Time taken to classify various open source projects for SQLI

<table>
<thead>
<tr>
<th>Repository Name</th>
<th>Number of Java Files</th>
<th>Total LLOC</th>
<th>Total Classification Time (Sec)</th>
<th>SQLI Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>gwtspeechbubble.googlecode.com</td>
<td>3</td>
<td>82,294</td>
<td>0.009</td>
<td>FALSE</td>
</tr>
<tr>
<td>ov2java.googlecode.com</td>
<td>4</td>
<td>133,122</td>
<td>0.010</td>
<td>FALSE</td>
</tr>
<tr>
<td>xmlui.googlecode.com</td>
<td>4</td>
<td>110,426</td>
<td>0.044</td>
<td>FALSE</td>
</tr>
<tr>
<td>permutationcombination.googlecode.com</td>
<td>12</td>
<td>327,806</td>
<td>0.043</td>
<td>FALSE</td>
</tr>
<tr>
<td>org2hash.googlecode.com</td>
<td>26</td>
<td>434,457</td>
<td>0.136</td>
<td>FALSE</td>
</tr>
<tr>
<td>teknoatolye.googlecode.com</td>
<td>39</td>
<td>666,449</td>
<td>0.147</td>
<td>FALSE</td>
</tr>
<tr>
<td>grimwepa.googlecode.com</td>
<td>43</td>
<td>1,315,067</td>
<td>0.668</td>
<td>TRUE</td>
</tr>
<tr>
<td>lambdacore.googlecode.com</td>
<td>56</td>
<td>969,340</td>
<td>0.177</td>
<td>FALSE</td>
</tr>
<tr>
<td>gracedm.googlecode.com</td>
<td>266</td>
<td>5,951,640</td>
<td>2.544</td>
<td>FALSE</td>
</tr>
<tr>
<td>oxygensoftwarelibrary.googlecode.com</td>
<td>545</td>
<td>4,534,779</td>
<td>7.173</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
8.3 Usability Study

In this section, we analyze the results of a user study\(^1\) that was conducted as part of the evaluation of our tool. The study followed the A/B testing format where participants used two tools to complete two tasks and provide feedback based on the experience they had while using both tools. FindBugs was chosen as the second tool due to its coverage in the literature [160, 161], its adoption by major companies such as Google [162], its open source nature, and its target language being Java.

First, the goal of the study is described, followed by the methodology, which includes a description of the participants, the apparatus and materials used, and the methods employed in the study. The results of the study are then presented along with a discussion of their significance.

8.3.1 Study Goal

The overall goal of the study was to ascertain the usefulness and usability of a recommender system in helping programmers write more secure code.

The answer to this question will provide a partial answer to the overarching goal of this dissertation as presented in Section 1.3.

8.3.2 Participants

Fourteen participants participated in the study (1 professor, 1 industry professional, 4 researchers, 3 undergraduate students, 1 master’s student, and 4 PhD students). Nine participants were in the age group 18–29, four between 30–49, and one between 50–64. Participants ranged in coding experience with 13 participants having at least 3 years experience and 1 participant between 0–2 years (See Figure 8.2). Participants were

\(^{1}\)IRB#:18-006
Figure 8.2: Participants’ coding experience

Figure 8.3: Participants’ primary programming languages

asked to select their primary programming languages and 9 participants selected Java and Python as their languages of choice. Figure 8.3 presents a summary of their language choices.
8.3.3 Apparatus and Materials

All participants used a Dell Latitude 3550 laptop (Intel Core i3 - 1.70GHz CPU, 64-bit, 8GB of RAM) to complete the tasks. The study took place in a classroom in the Harris Institute for Assured Information at Florida Institute of Technology, with one participant and one experimenter per interview; each session lasted 30 to 45 minutes. The Eclipse IDE (version Oxygen.3a 4.7.3) was installed on the computer beforehand. The VulIntel plugin and the FindBugs plugin (version 3.0.1) were also installed before the study started.

8.3.3.1 FindBugs Installation and Configuration

To install the FindBugs plugin for Eclipse, we followed the instructions presented on the FindBugs website\(^2\). This involved clicking Help on the Eclipse menubar and then selecting Install New Software. Next, we entered in the Work With input field the web address\(^3\) provided on the website. After following the on-screen instructions, the FindBugs plugin was installed and we were prompted to restart the Eclipse IDE. To have a fair comparison of tools, FindBugs was configured to report only security Bugs. This was done because FindBugs is able to find bugs related to bad practice, correctness, performance, etc, and we did not want unrelated issues to affect the scanning time or presentation of errors to the participants.

8.3.3.2 VulIntel Installation and Configuration

To install the VulIntel plugin, an Eclipse feature project (an installable plugin) was created and exported. A path to the plugin files on the file system was added to the Eclipse software installation system. This way the Work With input field could

\(^2\)http://findbugs.sourceforge.net/manual/eclipse.html

\(^3\)http://findbugs.cs.umd.edu/eclipse/
be populated with this location and similar on-screen installation instructions used for
FindBugs were displayed and followed.

The VulIntel plugin did not require any specific configurations. It presents all
information regarding vulnerabilities in one tab (VulIntel Plugin) and scans code for
vulnerabilities as the programmer types, as discussed in Section 7.4.

8.3.4 Methods

First, the experimenter presented the participant with an Informed Consent Form.
The experimenter reviewed the contents of the form and gave participants a randomly
assigned ID that was be used to identify the participant throughout the study. After
reviewing the contents of the consent form and the required tasks for the study, the
participant was given the option to withdraw or to proceed by signing the form. The
study then began with a short demographic-style questionnaire (See Appendix E.1) that
was designed using Google Forms. After signing the form, the interviewer told the
participant the order of the tools they would be using. Tool order was alternated to avoid
learning bias (i.e. 7 participants used FindBugs first before using VulIntel while 7 used
VulIntel before using FindBugs). The interviewer then explained to participants how to
use the first tool to scan their code for vulnerabilities and how to use the information
the tool provided to fix any potential vulnerabilities. Participants were told that they
should use only the information provided by the tool, and no other resources, to fix any
reported vulnerabilities.

Next, the experimenter activated screen-capturing (and audio-recording) software,
stepped aside, and allowed the participant to complete the two tasks using the first
tool. After completing the tasks using the first tool, the participant was then given
a questionnaire followed by an interview (See Appendix E.2 and E.3) based on their
experience using the tool to scan and fix the given code of potential vulnerabilities. If a
participant was unable to fix the vulnerabilities using the tool, the experimenter allowed
the participant to proceed with the next tool. The screen-capturing software was closed and the same experiment was given for the second tool.

### 8.3.4.1 Tasks

Each participant was given two tasks related to the top two taint-style vulnerabilities discussed earlier (See Section 6.2.2.1 and Section 6.2.2.2). Each task consisted of the user typing preselected sample code into the text editor of the Eclipse IDE while the code scanner window was open and the scanner activated. Two Java classes containing sample methods were created prior to the experiment with vulnerable portions of the code removed, so the participant could type, observe the behavior of the scanner, and use the information provided by the scanner to fix the vulnerability.

The code used for Task 1 (SQL Injection) is a modified version of an example provided by the Software Engineering Institute at the Carnegie Mellon University while the code used for Task 2 (Command Injection) was obtained from the OWASP website. These code samples with the vulnerable portion removed are provided in Appendix B and C, respectively. Additionally, the task sheet given to participants containing the vulnerable portions of code is provided in Appendix D.

### 8.3.5 Results and Analysis

Figure 8.4 provides a frequency summary of participants’ responses to four main questions asked on the questionnaire (See Appendix E) for each tool after participants completed the tasks. All four questions were presented using a 5-point Likert scale. As

---

4https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J+Prevent+SQL+injection

5https://www.owasp.org/index.php/Command_injection_in_Java

6“Strongly Disagree”, “Disagree”, “Neutral”, “Agree”, “Strongly Agree”

94
can be seen from the chart, more people agreed with VulIntel satisfying these questions positively than those who agreed that FindBugs did the same. If the Likert scale is collapsed into two categories (agree and disagree) by removing the neutral responses, the following can be concluded:

- 14 participants agreed that VulIntel provided helpful information including fixes for the two tasks given whereas only 1 participant agreed that FindBugs provided the same.

- 14 participants agreed that the VulIntel interface was usable whereas only 6 agreed that the FindBugs interface was usable.

- 13 participants indicated that they think VulIntel would help them write more secure code while only 4 participants think that FindBugs would help them to write more secure code.

- All participants stated that they would use the VulIntel system when coding while only 3 participants would use FindBugs.

In addition to the above questions, participants were asked whether it was easier for them to fix vulnerabilities based on a deeper understanding of the vulnerabilities rather than examples of other fixes. Interestingly, after completing tasks on the FindBugs system, 5 participants agreed that it is easier to fix vulnerabilities based on deeper understanding rather than using examples. After using the VulIntel system, 9 participants agreed that it was easier to fix vulnerabilities based on a deeper understanding of the vulnerabilities rather than using examples. These responses are rather odd given that the FindBugs system typically provides only a description of the bug/vulnerability whereas VulIntel provides a description in addition to example fixes. The responses to this question could be due in part to a misunderstanding of the question or because VulIntel provided both types of information.
Moreover, participants were asked whether they thought using the system in question would allow them to fix vulnerabilities faster than other tools. 3 participants responded positively (agreed) to this question regarding FindBugs while 9 responded positively regarding VulIntel. The responses to this question are not very meaningful since none of the participants has ever used code analyzers before.

Figure 8.4: Summary of participants’ responses to 4 main questions
8.3.5.1 Statistical Significance

It is of interest to determine any statistical significance regarding the study. Therefore, we first performed four paired sample T-tests for the four questions discussed previously. T-tests are used to determine whether the mean difference between two sets of observations is equal to zero. To obtain numeric data for carrying out the tests, the Likert scale was converted to an ordinal scale as follows: “Strongly Disagree”: 1, “Disagree”: 2, “Neutral”: 3, “Agree”: 4, “Strongly Agree”: 5. All T-tests were two-tailed and defined as follows: \( H_0 : \mu_d = 0 \) and \( H_1 : \mu_d \neq 0 \).

Further, to check whether the choices of participants depended on the order of the tools presented during the study, each T-test was accompanied by a two-way ANOVA test for each of the four main questions discussed earlier.

**Test 1 (Tool Helpfulness):** There is a difference between the level of help provided by FindBugs versus VulIntel in fixing vulnerabilities

Result: \( t = 10.3333, \ df = 13, \ p-value = 1.228e-07 \)

95% Confidence Interval = [1.751350, 2.677222]

With such a low p-value, we reject the null hypothesis.

The results of the ANOVA that tests whether there is interaction between the score assigned and the order of the tools as it relates to tool helpfulness is presented in Table 8.4. From the table, it can be seen that participants’ agreement is significant with a \( p-value \) of 1.62E-10. On the other hand, tool-order is not significant (\( p-value \) of 0.3) and interaction between tool-order*agreement is not significant (\( p-value=0.47 \)), indicating that participants’ agreement with the helpfulness of the tools did not depend on the order in which the tools were presented.
Table 8.4: Two-way ANOVA regarding tool helpfulness

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolOrder</td>
<td>1</td>
<td>0.321</td>
<td>0.321</td>
<td>0.9826</td>
<td>0.3315</td>
</tr>
<tr>
<td>agreement</td>
<td>1</td>
<td>36.619</td>
<td>36.619</td>
<td>111.9401</td>
<td>1.62E-10</td>
</tr>
<tr>
<td>ToolOrder:agreement</td>
<td>1</td>
<td>0.172</td>
<td>0.172</td>
<td>0.5269</td>
<td>0.4749</td>
</tr>
<tr>
<td>Residuals</td>
<td>24</td>
<td>7.851</td>
<td>0.327</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test 2 (Usability): There is a difference between the usability of the VulIntel interface versus the FindBugs interface.

Result: $t = 6.5655, df = 13, p-value = 1.81e-05$

95% Confidence Interval = [1.006424, 1.993576].

With such a low $p$-value, we reject the null hypothesis.

ANOVA results showing whether there is interaction between participants’ agreement and the order of the tools in the usability context are presented in Table 8.5. With a relatively low $p$-value (0.02) and significance level of 0.05, the tool-order factor was significant as it relates to usability. However, given the high $p$-value of 0.18, there is no interaction between tool-order and the usability rating assigned by participants for each tool.

Table 8.5: Two-way ANOVA regarding tool usability

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolOrder</td>
<td>1</td>
<td>1.75</td>
<td>1.75</td>
<td>5.8235</td>
<td>0.02381</td>
</tr>
<tr>
<td>agreement</td>
<td>1</td>
<td>23.7372</td>
<td>23.7372</td>
<td>78.991</td>
<td>4.67E-09</td>
</tr>
<tr>
<td>ToolOrder:agreement</td>
<td>1</td>
<td>0.5507</td>
<td>0.5507</td>
<td>1.8326</td>
<td>0.18844</td>
</tr>
<tr>
<td>Residuals</td>
<td>24</td>
<td>7.2121</td>
<td>0.3005</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test 3 (Ability of tools in helping users write more secure code): There is a difference between the views of participants regarding whether FindBugs or VulIntel will help them to write more secure code.

Result: $t = 7.3202, df = 13, p-value = 5.83e-06$
95% Confidence Interval = [1.309055, 2.405231]

With such a low p-value, we reject the null hypothesis.

ANOVA results showing whether there is interaction between participants’ agreement and the order of the tools in participants’ decision about the ability of each tool to help them write more secure code are shown in Table 8.6. These results show that tool-order had no impact on participants’ agreement.

Table 8.6: Two-way ANOVA regarding the ability of the tools in helping users write more secure code

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolOrder</td>
<td>1</td>
<td>0.571</td>
<td>0.571</td>
<td>1.6612</td>
<td>0.2097</td>
</tr>
<tr>
<td>agreement</td>
<td>1</td>
<td>33.549</td>
<td>33.549</td>
<td>97.5319</td>
<td>6.27E-10</td>
</tr>
<tr>
<td>ToolOrder:agreement</td>
<td>1</td>
<td>0.052</td>
<td>0.052</td>
<td>0.1524</td>
<td>0.6997</td>
</tr>
<tr>
<td>Residuals</td>
<td>24</td>
<td>8.256</td>
<td>0.344</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test 4 (Tool adoption): There is a difference in participants’ preference in using VulIntel versus FindBugs when coding

Result: \( t = 6.1085, \ df = 13, \ p-value = 3.729e-05 \)

95% Confidence Interval = [1.338837, 2.804020]

With such a low p-value, we reject the null hypothesis.

Table 8.7 shows the results of a two-way ANOVA test that determines whether tool-order interacted with participants’ agreement. The results show that tool-order may have played an insignificant \( p-value=0.09 \) and significance=0.05 role in which tool participants would prefer to use for code analysis.
Table 8.7: Two-way ANOVA regarding tool adoption

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolOrder</td>
<td>1</td>
<td>0.893</td>
<td>0.893</td>
<td>2.0344</td>
<td>0.1666</td>
</tr>
<tr>
<td>agreement</td>
<td>1</td>
<td>38.194</td>
<td>38.194</td>
<td>87.025</td>
<td>1.88E-09</td>
</tr>
<tr>
<td>ToolOrder:agreement</td>
<td>1</td>
<td>1.344</td>
<td>1.344</td>
<td>3.0618</td>
<td>0.0929</td>
</tr>
<tr>
<td>Residuals</td>
<td>24</td>
<td>10.533</td>
<td>0.439</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8.3.6 Study Limitations

Several possible study limitations should be taken into consideration when assessing or generalizing the results of this experiment. The main limitations are as follows:

8.3.6.1 Sample Size

The number of participants, though relatively diverse, was small ($N = 14$). There is the potential of obtaining different results with a larger sample. However, since it is typical in the usability community to conduct studies with focus groups between 6-10 participants, the results presented here are acceptable. Further, the statistical significance reported helps to strengthen the conclusions.

8.3.6.2 Gap between Tool Age

The gap between the age of both tools is significant. FindBugs was originally released in 2006, with its most recent release in 2015. While it has gone through many stages of enhancements, its focus may be for a different generation of users and programmers. The proposed tool in this study has not yet been released to the public and was designed specifically to address the problem identified in this thesis (See Section 1.2). Therefore, the age difference between the two tools may have some effect on the results.
8.3.6.3 Experimenter Demand Effects (EDEs)

Demand effects refer to bias stemming from participants inferring the purpose of an experiment and tailoring their responses to help confirm a researcher’s hypothesis [163]. We believe this limitation was partially overcome by the fact that none of the participants has ever seen or worked with the tools featured in the study before. Participants required explanations from the experimenter on how to use the tools. Additionally, at least 5 of the participants have never met the experimenter beforehand. Finally, the level of honesty observed in participants’ subjective responses as to whether they liked or would use the tools featured in the study, as discussed above, show that EDEs may be very minimal in this study.

8.3.6.4 Learning Effects

Learning effects, which are due to the order of presentation, might have been a limitation. However, we overcame this effect by alternating the order in which participants evaluated each tool.
Chapter 9

Conclusions

Vulnerabilities in program code continue to play a significant role in system failures and data breaches, costing individuals and organizations large sums of money and other immeasurable damages. While code reviews are strongly recommended to mitigate software errors, they are not feasible for all projects. Therefore, the process of finding and correcting mistakes in program code should be automated and employed as soon as the first key is pressed on a keyboard during development.

In this dissertation, a methodology is proposed, designed and evaluated to help programmers avoid vulnerabilities as they type code during development. The proposed methodology advocates the use of text mining and machine learning to extract features from code repositories in order to train classifiers to detect vulnerabilities, thereby classifying program code as safe or unsafe based on a knowledge base.
9.1 Summary of Contributions

This research featured the design and evaluation of a recommender system that helps programmers find and mitigate vulnerabilities in their programs during development. The research addressed the following research question:

How applicable are machine learning and text mining in creating a recommender system that provides live security advice to programmers that will improve the security of the program being developed?

A usability study showed that all 14 participants involved agreed that the proposed system was more usable than the FindBugs system, and it provided more helpful advice including fixes for the tasks they completed using the system. In addition, all but 1 participant indicated that the proposed system would help them to write more secure code. The results were statistically evaluated and paired sample T-tests and two-way ANOVA suggest that there is statistical significance.

The following is a summary of the contributions made in this dissertation:

1. Developed a methodology that uses data from the National Vulnerability Database and open source projects to detect vulnerabilities in source code

2. Designed a MapReduce algorithm, set up a distributed cluster, and used Apache Hadoop to process a large code base of over 1.6 million Java files (over 7000 open source project) to extract features for detecting and mitigating vulnerabilities

3. Extracted insights from the literature to propose a usable interface for the system and conducted formative testing of over 100 participants from over 5 countries to elicit knowledge on the system design

4. Utilized responses from the knowledge elicitation survey to implement the proposed system for detecting two taint-style vulnerabilities
5. Ported the system to an Eclipse plugin, conducted usability study in the form of A/B testing to determine the usefulness and usability of the system, and used the results to demonstrate that the system can help programmers write more secure code, thereby answering the research question.

9.2 Recommended Future Work

Future directions for this work include the following:

1. The use of deep learning to determine the features for detecting vulnerabilities instead of using hand-coded features.

2. Expanding the work by detecting and correcting more vulnerabilities/weaknesses in the SANS/CWE 2011 list of Most Dangerous Software Errors. The analysis in this work showed that by correcting the two featured vulnerabilities, 1,300 out of 1,500 vulnerabilities in the 2017 NVD release could be avoided.

3. Improving the user interface based on the responses received from participants in the usability study.

4. Expanding the tool to support more programming languages and IDEs.
Bibliography


105


Appendix A

Knowledge Elicitation Survey

Included in this section is a copy of the knowledge elicitation survey that was conducted to determine the current use of code analyzers among programmers and to solicit their views on the design of our proposed system that uses IntelliSense technology to detect and mitigate vulnerabilities.

A.1 General Questions

1. What is your occupation?

- [ ] Undergraduate student
- [ ] Graduate Student
- [ ] Professor
- [ ] Industry Expert
- [ ] Freelancer
- [ ] Other
2. How would you describe your level of familiarity with the following programming languages?

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>Somewhat familiar</th>
<th>Familiar</th>
<th>Very familiar</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>C#</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>C</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>C++</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Python</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>JavaScript</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>PHP</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Perl</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

3. Please indicate your familiarity with the following source code editors and/or integrated development environments (IDEs)

<table>
<thead>
<tr>
<th></th>
<th>Have heard of but never used</th>
<th>Have used</th>
<th>Never heard of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Netbeans</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>IntelliJ IDEA</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Emacs</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Vi/Vim</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Other (please specify)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. How important or unimportant is it for you to develop secure code?

- Very important
- Important
- Unimportant
- Very unimportant

5. How do you currently scan your code for weaknesses (vulnerabilities) or unsafe practices?

- I use a static analyzer such as FindBugs
- I use a dynamic analyzer such as Java PathFinder
- I use both static and dynamic analyzers
- I write code and another party scans it for vulnerabilities
- I currently do not scan my code for vulnerabilities
- Other ____________

A.2 Rate Existing Scanners

1. If you answered (a) - (c) in question 5, how helpful do you find the warnings/advice provided by the selected scanner?

- The advice is very helpful in fixing vulnerabilities
- The advice provided does not help in fixing vulnerabilities
- The advice given is somewhat helpful in fixing vulnerabilities
A.3 Intellisensing Vulnerabilities

1. Do you currently utilize IntelliSense technology (also known as “code completion” or “code hinting” during coding?
   - Yes
   - No

2. What is your opinion about detecting vulnerabilities using IntelliSense technology?
   - I would appreciate a system that can scan my code for vulnerabilities as I code
   - I do not care about such technology, but I believe other programmers would appreciate a system that utilizes this technology
   - I do not think this would be a good idea

A.4 Knowledge Elicitation

We will now show you a mockup of a tool we are developing that is designed to help programmers find and fix vulnerabilities as they code. The tool will be created using machine learning techniques and implemented as a plugin in common IDEs such as Eclipse.

1. We will utilize triggers (e.g.: key-press combinations, IntelliSense) to initiate the scanner, which will run in the background to scan for vulnerabilities.

2. If vulnerabilities are found, a list will be populated with the common weakness ID.

3. Clicking a weakness ID in the list will present a brief overview of the vulnerability and display a ranked set of examples of how people fix the vulnerability.
4. Clicking on an example will present the user with sample code that the user may use as a reference to fix the vulnerability.
1. Consider situations where you are writing code. In what situations would you utilize this plugin?

☐ Before code release  ☐ When I finish a module
☐ During a nightly build  ☐ When I finish a class
☐ As I type code  ☐ Other __________

2. What do you like or dislike about the plugin featured in the mockup?

______________________________________________________________________________

3. For what types of project would you use this plugin?

☐ work projects  ☐ Fun projects
☐ school projects  ☐ Open source projects
☐ While freelancing  ☐ Other __________

4. What types of vulnerabilities, if any, would you like to be able to detect with this software/plugin?

______________________________________________________________________________

5. If you have any further comments or suggestions, you may enter them below.

______________________________________________________________________________

______________________________________________________________________________
Appendix B

SQLI Code Used In Usability Study

Listing B.1: Sample Code Used for Task 1 in User Study

```java
package dissertation_evaluation;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

/**
 * This is a sample class for to evaluate two vulnerability scanners. It is a
 * piece of code that could be used by an authentication system to authenticate
 * users. You are not expected to run the program.
 */
```
class Login {
    
    public static Connection getConnection() throws SQLException, ClassNotFoundException {
        String dbConnection = null;
        Class.forName("org.sqlite.JDBC");
        dbConnection = DriverManager.getConnection("jdbc:sqlite:").toString();
        return DriverManager.getConnection(dbConnection);
    }

    /**
     * Function to create secure hash of password
     *
     * @param password
     * @return hashed password
     */
    static String hashPassword(char[] password) {
        // Create hash of password to secure it
        return password.toString();
    }

    /**
     * This method authenticates users
     * @param username
     * @param password
     * @return
     * @throws SQLException
     * @throws ClassNotFoundException
     */
    
}
public static boolean doLogin(String username, char[] password) throws SQLException, ClassNotFoundException {
    Connection connection = getConnection();
    if (connection == null) {
        // Handle error
    }
    try {
        // ENTER THE CODE PROVIDED BY THE EXPERIMENTER HERE
        // ENTER THE CODE PROVIDED BY THE EXPERIMENTER HERE
    } finally {
        try {
            connection.close();
            return false;
        } catch (SQLException x) {
            // Forward to handler
        }
    }
}

/**
 * The main method to invoke the system
 *
 * @param args
 * @throws SQLException
 * @throws ClassNotFoundException
 */
public static void main(String[] args) throws ClassNotFoundException, SQLException {

    Scanner scan = new Scanner(System.in);
    String username;
    char[] password;
    boolean loggedIn = false;

    System.out.println("Enter Username: ");
    username = scan.next();

    System.out.println();

    System.out.println("Enter password: ");
    password = scan.next().toCharArray();

    loggedIn = doLogin(username, password);

    if (loggedIn)
        System.out.println("You are successfully logged in.");
    else
        System.out.println("Incorrect credentials provided");
}
Appendix C

Command Injection Code Used In Usability Study

Listing C.1: Sample Code Used for Task 2 in User Study

```java
package dissertation_evaluation;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

/**
 * This is a sample class to evaluate two vulnerability scanners. It is a piece of code that could be used to execute the cmd.exe program in Microsoft Windows.
 * You are not expected to run the program.
 */
public class CodeExecutor {
```

135
public static void main(String[] args) throws IOException {
    if(args.length != 1) {
        System.out.println("No arguments");
        System.exit(1);
    }

    // ######################################
    // ENTER THE CODE PROVIDED BY THE EXPERIMENTER HERE
    // ######################################

    InputStream is = proc.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
}
Appendix D

Usability Study Tasks

This section contains the list of tasks given to participants who participated in the usability study.

D.1 Overview

This experiment is part of a dissertation that features the development and evaluation of a recommender system for improving program security through source code mining and knowledge extraction. The system is intended to be used by Java programmers to help find vulnerabilities in program code and provide advice to programmers to help them mitigate the vulnerabilities and make their programs more secure.

In this evaluation, you will complete two tasks on two separate systems: FindBugs and VulIntel. The experimenter will tell you which system to use first.
D.1.1 FindBugs

FindBugs is a lightweight code analyzer that uses bug patterns or rules to find bugs in program code and present recommendations to programmers to help them fix potential bugs. When using the FindBugs system, there should be 3 FindBugs tabs open, namely Bug Explorer, Bug Info, and Bug Reviews (If these tabs are not open, click on Window → Show View → Other → FindBugs and open each one). Additionally, we have configured FindBugs to report only security bugs. It was also configured to run automatically. To validate these settings, right-click on the dissertation_evaluation project, click Properties → FindBugs.

D.1.2 VulIntel

VulIntel is a recommender system that uses machine learning to find vulnerabilities in program code and offers recommendations to programmers to help them fix potential vulnerabilities. When using the VulIntel system, there should be a tab labeled VulIntel Plugin located below the Eclipse Code editor. (If this tab is not open, click on Window → Show View → Other → VulIntel → VulIntel Plugin)
D.2 General Demographic Questions

First, the experimenter will provide you with a link to a demographic form for you to complete before starting the experiment. The form will collect information such as your occupation, age group, years of coding experience, etc.

D.3 How to Use the Tools

D.3.1 FindBugs

If you are told to use the FindBugs system when completing the experiment and no warnings are displayed, right-click on the Project dissertation_evaluation in the Eclipse Project Explorer. Choose FindBugs → Find Bugs.

Read any information provided in the Bug Explorer, Bug Info, and Bug Reviews tabs to check for bugs/vulnerabilities. Use the information provided, if any, to fix the reported bugs/vulnerabilities.

After completing both tasks, inform the experimenter when you are finished. You will then complete a questionnaire and answer a few questions about your experience.

D.3.2 VulIntel

If you are told to use the VulIntel system, follow the information displayed in the user interface to fix the reported bugs/vulnerabilities. If the tool alerts you of a potential vulnerability, please read the information in the Quick Overview panel and browse the example fixes provided in the How People Fix It panel. Please try to fix the vulnerability using the examples provided and observe the VulIntel Window.

After completing both tasks, inform the experimenter when you are finished. You will then complete a questionnaire and answer a few questions.
D.4 TASK 1: SQL Injection

The experimenter should have a Java Project called dissertation_evaluation open in the Eclipse IDE.

1. Please open the Login.java class file and locate the section labeled ENTER THE CODE PROVIDED BY THE EXPERIMENTER HERE.

2. Type the code provided below in Listing D.1 in the Eclipse Code editor.

Listing D.1: Code Vulnerable to SQL Injection

```java
String pwd = password.toString();

String sqlString = "SELECT * FROM db_user WHERE username = \\
                    " + username + ", AND password = " + pwd + ";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sqlString);
connection.close();

if (!rs.next()) {
    throw new SecurityException("User name or password incorrect");
}

// Authenticated; proceed
return true;
```
D.5 TASK 2: Command Injection

1. Please open the CodeExecutor.java class file and locate the section labeled ENTER THE CODE PROVIDED BY THE EXPERIMENTER HERE.

2. Type the code provided below in Listing D.2 in the Eclipse Code editor.

Listing D.2: Code Vulnerable to Command Injection

```java
Runtime runtime = Runtime.getRuntime();
String[] cmd = new String[3];
cmd[0] = "cmd.exe";
cmd[1] = "/C";
cmd[2] = "dir " + args[0];
Process proc = runtime.exec(cmd);
```
Appendix E

Usability Study Questions

This section contains the list of questions given to participants who participated in the usability study.

E.1 General Demographic Questions

The following questions were given to participants before they started the study:

1. What is your age group?
   - ○ 18-29 years old
   - ○ 30-49 years old
   - ○ 50-64 years old
   - ○ 65 years and over

2. What is your occupation?
3. Select your primary programming languages

- [ ] Java
- [ ] C#
- [ ] Visual Basic
- [ ] C
- [ ] C++
- [ ] Python
- [ ] Javascript
- [ ] PHP
- [ ] Perl
- [ ] Other __________

4. How many years of coding experience do you have?

- [ ] 0-2
- [ ] 3-5
- [ ] 6-8
- [ ] 9-11
- [ ] 12-14
- [ ] 15-20
- [ ] Over 20 years

**E.2 Post-Task Completion Questionnaire**

The following questionnaire was given to participants after completing the tasks for one tool:

1. ______ provided me with helpful information including examples on how to fix vulnerabilities

   - [ ] Strongly agree
   - [ ] Agree
   - [ ] Neutral
   - [ ] Disagree
   - [ ] Strongly disagree
2. Overall, the ______ interface was usable

○ Strongly agree  ○ Disagree
○ Agree  ○ Strongly disagree
○ Neutral

3. I think the ______ system will help me to write more secure code

○ Strongly agree  ○ Disagree
○ Agree  ○ Strongly disagree
○ Neutral

4. I would use the ______ system when coding

○ Strongly agree  ○ Disagree
○ Agree  ○ Strongly disagree
○ Neutral

5. I think using the ______ system will allow me to fix vulnerabilities faster than other tools

○ Strongly agree  ○ Disagree
○ Agree  ○ Strongly disagree
○ Neutral
6. It is easier for me to fix vulnerabilities based on a deeper understanding of the vulnerabilities rather than examples of other fixes.

- Strongly agree
- Agree
- Neutral
- Disagree
- Strongly disagree

E.3 Post-Task Completion Interview

1. Were you able to complete all the tasks given to you on the _____ system? Why or why not?

________________________________________________________________________

2. What did you like about using the code analyzer on the _____ system?

________________________________________________________________________

3. What did you dislike about using the code analyzer on the _____ system?

________________________________________________________________________
Appendix F

List of Publications

Below is a list of publications that have resulted from the research conducted to produce this dissertation:

2018 extracting knowledge from open source projects to improve program security (fitzroy nembhard, marco carvalho and thomas eskridge), in southeast con 2018, ieee, 2018.


the following paper related to this research is under peer review:

2018 helping programmers fix vulnerabilities using intellisense technology (fitzroy nembhard, marco carvalho and thomas eskridge), submitted to annual computer security applications conference (acsac’18).