CLASSIFYING AND PREDICTING SOFTWARE SECURITY VULNERABILITIES BASED ON REPRODUCIBILITY

by

MARYAM DAVARI

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
December 2016

Copyright © Maryam Davari, 2016
Abstract

Security defects are common in large software systems because of their size and complexity. Although efficient development processes, testing, and maintenance policies are applied to software systems, there are still a large number of vulnerabilities that can remain, despite these measures.

Some vulnerabilities stay in a system from one release to the next one because they cannot be easily reproduced through testing. These vulnerabilities endanger security of the systems. We propose vulnerability classification and prediction frameworks based on vulnerability reproducibility. The frameworks are effective to identify the types and locations of vulnerabilities in the earlier stage, and improve the security of software in the next versions (referred to as releases).

We expand an existing concept of software bug classification to vulnerability classification (easily reproducible and hard to reproduce) to develop a classification framework for differentiating between these vulnerabilities based on code fixes and textual reports. We then investigate the potential correlations between the vulnerability categories and the classical software metrics and some other runtime environmental factors of reproducibility to develop a vulnerability prediction framework.

The classification and prediction frameworks help developers adopt corresponding mitigation or elimination actions and develop appropriate test cases. Also, the
vulnerability prediction framework is of great help for security experts focus their effort on the top-ranked vulnerability-prone files. As a result, the frameworks decrease the number of attacks that exploit security vulnerabilities in the next versions of the software.

To build the classification and prediction frameworks, different machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are employed. The effectiveness of the proposed frameworks is assessed based on collected software security defects of Mozilla Firefox.
Acknowledgments

First, I offer my gratitude to my supervisor, Prof. Mohammad Zulkernine for his support, encouragement, and motivation throughout my thesis.

I thank my parents, brother, and lovely sister who provide me with all supports and confidence.

I also like to thank the Canada Research Chairs (CRC) and the Mitacs which partially funded this research.
Contents

Abstract i

Acknowledgments iii

Contents iv

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1
  1.1 Motivation ........................................... 1
  1.2 Overview ............................................. 3
  1.3 Contributions ...................................... 6
  1.4 Organization of Thesis .............................. 7

Chapter 2: Background and Related work 8
  2.1 Terminology .......................................... 8
  2.2 Software Defect Classification Based on Reproducibility ........................................... 9
    2.2.1 Mandelbug (MB) ................................ 10
    2.2.2 Bohrbug (BB) .................................. 12
  2.3 Overview of Machine Learning Techniques ......................................................... 13
    2.3.1 C4.5 Decision Tree ................................ 14
    2.3.2 Random Forests ................................... 14
    2.3.3 Logistic Regression ................................. 15
    2.3.4 Naive Bayes ....................................... 16
  2.4 Related Work ......................................... 17
    2.4.1 Software Vulnerability Classification ................. 17
    2.4.2 BB and MB Classification ........................... 18
    2.4.3 Software Vulnerability Prediction ...................... 19
  2.5 Challenges in Vulnerability Study .......................................................... 21
  2.6 Summary ............................................... 21
# List of Tables

3.1 Example of different levels of severity in Bugzilla [1] . . . . . . . . . . 31
3.2 Example of BV and MV . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.3 Distribution of BVs and MVs . . . . . . . . . . . . . . . . . . . . . . 47
3.4 Distribution of BVs and MVs for different categories in the data set . 47
3.5 Top discriminative features . . . . . . . . . . . . . . . . . . . . . . . . 49
3.6 Comparison between different classifiers . . . . . . . . . . . . . . . . . 54

4.1 Software complexity metrics . . . . . . . . . . . . . . . . . . . . . . . 59
4.2 Factors based on the internal environment of the system . . . . . . . 62
4.3 Dimensionality reduction by PCA for BV . . . . . . . . . . . . . . . . . 70
4.4 Dimensionality reduction by PCA for MV . . . . . . . . . . . . . . . . 70
4.5 Metric selection based on the regression models with p-value < 0.05 . 71
4.6 Comparison of different machine learning techniques for the MV-prone
    prediction framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.7 Pearson correlation for the hypotheses $H1$, $H2$, and $H3$ . . . . . . . 80
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of the classification and prediction frameworks</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>An overview of the proposed vulnerability classification framework</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Snapshot of a log message</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Snapshot of file extraction from command line using <code>hg</code></td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>A textual report sample of Bugzilla</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>An overview of the vulnerability collection approach.</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>A vulnerability report in MFSA</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>An overview of the correlated metrics investigation for BVs and MVs</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Metric Extractor component</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Screen plot of PCA.</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>An overview of the proposed MV-prone prediction framework</td>
<td>73</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

As software systems are becoming larger and more complicated, software defects (referred to as faults or bugs) are inevitable. They lead to system failures. A specific type of defect which can enable unauthorized users to break security policies of a system is known as a vulnerability or security hole. Vulnerabilities are a big threat to the security of information systems. For instance, a vulnerability in China Software Developer Network (CSDN), the largest Chinese network for software developers, caused the information leakage of 6 million users in 2011 [2]. Furthermore, a flame virus attacked Middle East computers with Microsoft operating systems in 2012. It used the digital signature spoofing vulnerability of operating systems to conceal itself [3].

Security faults incur further expenses and serious damages than other fault types. For instance, on average, more than $220 million losses due to cyber-attacks are reported annually [4]. It is difficult to detect a vulnerability before it manifests itself as a security failure in the operational stage as security is not seriously considered
1.1. MOTIVATION

during all of the software development life cycle.

Classification is an effective approach to improve the quality of software systems. Classification makes abstraction and generalization of the characteristics of security problems in a system. Its results can improve the quality of all phases of a software development process. Vulnerability classification involves remarkable manual efforts, and many corporations do not have enough budget for it. In addition, new vulnerabilities are being discovered rapidly. There is an urgent need for an automatic vulnerability type classification.

We analyse a large number of vulnerability reports in a security issue tracker (Mozilla Foundation Security Advisory (MFSA) [5]), and understand that developers spend a significant amount of time, effort, and budget in the testing phase to fix some security issues. Some of them are ignored because the security failures cannot be reproduced. Therefore, vulnerabilities with complicated activation conditions stay in a system and endanger its security. This situation enables attackers to exploit vulnerabilities in the next versions based on the identified vulnerabilities in the previous versions.

One motivation behind this research is to classify security vulnerabilities based on their reproducibility. Identifying vulnerabilities with hard reproduction process enables large corporations to deliver secure applications within acceptable cost and time. Fixing a defect in the testing phase is nine times cheaper than the operational stage [6]. Based on the reproducibility of security issues, appropriate countermeasures (e.g., test cases and quality assurance mechanisms) can be designed. For example, when a vulnerability is labelled as a hard to reproduce one, it informs developers that the classical testing techniques cannot be effective to reproduce the vulnerability.
Special strategies should be employed. It can dramatically decrease the number of security failures in the operational phase.

Furthermore, developing secure software is complex, costly, and time-consuming, and many corporations are not capable of doing it. One efficient solution is to predict vulnerability-prone parts of systems that is another motivation of this thesis. A vulnerability prediction model identifies program units (e.g., modules, files, or functions) which are more sensitive to attacks. This model is vital for web applications because non-secure web applications can enable attackers to steal user information. Throughout this thesis, vulnerabilities refer to post-release vulnerabilities (vulnerabilities that cause security failures after releases).

1.2 Overview

Analysing software defects with respect to their reproducibility is proposed by Grottke et al. [7]. They introduce two definitions in terms of failures: Bohrbug (BB) (i.e., an easily reproducible failure in the testing phase) and Mandelbug (MB) (i.e., a hard to reproduce failure through the testing phase). We adopt the BB and MB terminologies for security vulnerabilities. In this thesis, BV and MV represent security vulnerabilities with BB and MB characteristics, respectively.

Based on these concepts, we propose two frameworks: one for vulnerability classification and one for vulnerability prediction. The vulnerability classification framework classifies vulnerabilities into the BV and MV categories and the prediction framework anticipates the MV-prone files.

An overview of the frameworks are shown in Figure 1.1. The classification framework uses different resources: textual report on BVs and MVs, code change history
1.2. OVERVIEW

Figure 1.1: Overview of the classification and prediction frameworks

(in revision control), and source code. The Feature Extractor component extracts different features from the resources. After integrating the code history and textual reports, this component builds a repository of risky files (i.e., files that have been changed to fix security failures) to calculate the metrics. Then, the Classifier Builder component builds a classifier by applying different well-known machine learning techniques (including C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) on the collected feature values. When an unknown vulnerability is introduced into the classification framework, the Category Identifier recognizes its category (BV or MV) by using the classifier.

Early identification of vulnerability-prone files is another objective of this thesis. Detecting and fixing vulnerabilities are expensive processes in the software life cycle
1.2. OVERVIEW

(SLC), and their costs increase with time. Hence, we build a vulnerability prediction framework. We hypothesize that the complicated code can decrease the probability of vulnerability reproducibility. Furthermore, previous work has shown the effectiveness of the software metrics for fault prediction [8–11]. Therefore, we study the classical software metrics for the vulnerability prediction.

We also introduce a set of attributes related to the environment of the system which include concurrency, interaction of the program with system states, runtime errors, and memory management (predominant vulnerabilities in our data set are related to the memory). We assume that they can increase the probability of vulnerabilities with hard reproduction. The Metric Extractor component extracts the metric values for the risky files. Then, the Metric Analyzer investigates the linear and multiple linear correlations between the metrics and the target categories. It identifies the most correlated metrics with the BV and MV categories.

As MVs cannot be easily detected through the testing phase, the Predictor Builder component builds a predictor by applying the machine learning algorithms (namely Random Forest, C4.5 Decision Tress, Logistic Regression, and Naive Bayes) on the most correlated metrics. If the category of a new arrival vulnerability is identified as MV by the classification framework, the relevant vulnerable files are passed to the MV-prone predictor to predict if the involved risky files are MV-prone or non MV-prone files.

MV-prone file identification can dramatically decrease the total number of MVs in the project. Vulnerable files most likely will introduce more vulnerabilities in the future versions. Hence, instead of investigating the whole project, developer effort is concentrated on the top-ranked files. This approach helps develop cost-effective
1.3 Contributions

The main contributions of this research are as follows:

1. Propose a software vulnerability classification framework based on the reproducibility of security failures. It classifies security issues based on the textual reports and code fixes. Understanding the type of software vulnerabilities from a reproducibility perspective helps developers design appropriate corrective actions (the most costly part of the development and maintenance phases). For example, when a vulnerability is labelled as an MV, it informs developers that the classical testing techniques cannot be effective.

2. Propose a prediction framework to automatically predict the vulnerability-prone files [13]. Fixing MV-prone files in the earlier stages can dramatically decrease the total number of MVs in the project because vulnerable files more likely will introduce more vulnerabilities in the future versions. In addition, it is a great help for security risk managements focus and prioritize their security effort with respect to the most vulnerable files. This model helps develop cost-effective secure software [12]. Different machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are employed to build an MV prediction model with high precision and accuracy.

3. Empirically analyse actual vulnerabilities of Mozilla Firefox to evaluate the effectiveness of the classification and prediction frameworks. The classification framework identifies the relevant risky files that are really beneficial for doing secure software [12].
1.4 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 begins with basic terminologies. Then, defect classification techniques based on reproducibility are reviewed. The used machine learning techniques for the vulnerability classification and prediction are briefly explained. Also, previous work related to the vulnerability classification and prediction is compared and contrasted with each other. In Chapter 3, the classification framework with its components are described in detail. It explains the procedure of vulnerability and feature extractions. Then, several statistical classifiers are constructed based on the machine learning techniques, and the results are discussed. Chapter 4 provides an overview of the software metrics. Then, the correlation between the considered metrics and BVs and MVs are investigated. Due to the complicated reproduction of MVs, an MV-prone prediction framework is developed. Similar to the classification framework, different machine learning techniques are used to construct predictors and the results are compared. Chapter 5 summarizes the research along with the limitation of the work. It concludes with the outline of future work.
Chapter 2

Background and Related work

This chapter describes the basic terminology and background knowledge that is required to comprehend the proposed frameworks in this thesis. Then, software defect classification techniques based on the reproducibility are presented. This chapter briefly reviews different machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) that are employed to build classifiers and predictors in this research. The related research on software defect classification, vulnerability classification, and vulnerability prediction techniques are compared and contrasted. This chapter also states the challenges of studying software vulnerabilities.

2.1 Terminology

In this section, the definition of some of the widely used terms in this dissertation are described based on the [IEEE-1990] standard [14]:

**Fault**: A fault is a wrong process, step, or data in a computer program the execution of which may lead to an error. The terms 'bug' and 'defect' express the same meaning.

**Error**: An error means a measured value that differs from a determined or correct
value. For example, a correct result is 10, while the computed value is 13.

**Failure:** When a system is not able to function according to the specified requirements, a software failure happens.

**Vulnerability:** A software vulnerability is a defect in the specification, design, development, maintenance, or configuration of a software system. It can enable unauthorized users to break the security policy of the system. It is also known as security hole.

**Security policy:** Security policy divides the states of a system into the authorized (secure) and unauthorized (non-secure) states. Security policy is the statement of what is allowable from the security perspective. A secure system does not experience unauthorized states. It starts from an authorized state and ends in an authorized state.

**Security failure:** The exploitation of a vulnerability leads to a security failure which breaks the security policy implicitly or explicitly [15]. Vulnerabilities and security failures are sub-categories of faults and failures, respectively [16, 17].

Despite the importance of analysing security vulnerabilities from a reproducibility perspective, there is no significant effort in the literature in this direction of research. In the next section, we analyse defect classification strategies to get a better idea about the issues that can influence the reproducibility of security vulnerabilities. Those research works were the main motivation behind our research.

### 2.2 Software Defect Classification Based on Reproducibility

Analysing software defects from a reproducibility perspective was first discussed by Gray [18] in classifying reported failures of Tandem software products. The main
2.2. SOFTWARE DEFECT CLASSIFICATION BASED ON REPRODUCIBILITY

goal of his classification was to evaluate the effectiveness of implementations of fault tolerance. He observed that sometimes, a task does not fail even though it has failed in the past. Gray assumes that a fault can be classified into two categories: hard bug\(^1\) and soft bug.

The hard bug is also named Bohrbug (BB) or solid bug. The failure caused by a hard bug is easily reproduced. BB is described in following subsections in detail. The soft bug is also named elusive bug or Heisenbug. Its name alludes to Werner Karl Heisenberg, the author of the uncertainty principle. The soft bug has inconsistent manifestation due to its dependency on the environmental conditions.

Gray believes that hard bugs can be discovered and removed via testing and debugging techniques, while Heisenbugs cannot be detected always. Some research [19, 20] demonstrates that this assertion cannot be correct always. Grottke et al. [21] work on the fault classification proposed by Gray [18], and they correct that classification mechanism according to Lindsay's definition for Heisenbug [22]. They proposed a new classification scheme in 2005. Based on this classification, software defects can be classified into two distinct categories Bohrbug (BB) and Mandelbug (MB) by considering additional factors (such as behaviour, cause, and manifestation). In the following subsections, MB and BB are described\(^2\), respectively.

### 2.2.1 Mandelbug (MB)

Even though there are direct relationships between the fault, error, and failure (reviewed in Section 2.1), sometimes, some defects exhibit different and chaotic behaviour under seemingly similar conditions due to the following reasons [23]:

\(^1\)The definition of the hard bug is different from the hard to reproduce vulnerabilities in this research.

\(^2\)MB is presented first to facilitate the discussion of BB based on MB.
1. Inconsistent activation conditions: It is beneficial to understand the relationship between static defects and dynamic failure occurrences. Executing a part of a system may not lead to an immediate failure, but it deviates from the correct internal system condition.

2. The effect of the internal environment of the system on activation conditions: The states of a system (e.g., hardware, operating system behaviour, and application) executing the program can impact defect activation. The activation of defects is influenced by states of the system running the program including hardware, software (e.g., operating system and other application programs running on the system), etc.

3. The time lag between defect activation and failure occurrence: During the time difference, various error states are experienced. Consequently, detecting the user action which activates the fault is difficult. Trying to reproduce the failure might not be successful in all circumstances.

4. Error propagation: A security failure caused by a vulnerability in a method is manifested, while the method has been used earlier several times without any failure. The reproduction of this failure requires analysing previous states and subsystems through error propagation. For instance, if an algorithm is not correctly implemented, it will produce faulty values for some program variables. The wrong result(s) might be used by other parts of the program. One error introduces more errors into the system. When one of the incorrect results which can influence the system behaviour is used, the system experiences a failure. Briefly, it can be stated that the transformation of fault, error, and failure is called error propagation. All the reasons are related to the activation conditions.

A defect with the above mentioned features is called Mandelbug (MB). The term
alludes to ‘Mandelbrot set’. For instance, a computation error in a function might stay in the memory without any immediate deviation from the correct service. Only, the failure is experienced when the result is used. This is an example of MB with a delay between fault activation and failure occurrence.

Eliminating MBs before software release is not an easy task. Reproducing a failure due to a MB is difficult even if its underlying reason is known. For instance, in a multi-threaded program, if threads are not sufficiently synchronized, a race condition might happen. Reproducing a failure that happens because of running threads in a specific order is really difficult since threads are scheduled by the operating system. The programmer needs to predict the order of thread execution that caused the race condition and then fix the issue. To recover a failed system because of an MB, rebooting or restarting the system might be effective since it makes some system states clean [23]. MBs require accurate consideration and special techniques should be employed to deal with them because they do not have consistent behaviour. They remain in the system without any manifestation, maintaining the capability to cause severe damages.

2.2.2 Bohrbug (BB)

BB alludes to physicist Niels Bohr and his simple and intelligent atom model. A BB is an error in a program based on inputs. A security failure caused by a BB is always reproducible because its activation and error propagation are not related to the reasons discussed for MBs in the previous subsection. For instance, if the system crashes with the null pointer access, it always crashes with the given input. Due to the deterministic behaviour of BBs, it is expected to be detected and fixed easily by
the testing and debugging techniques. It is assumed that the number of BBs should be lower than the number of MBs in software systems. Recovering or restarting a failed system due to a BB may not work as the input is the reason of a failure [23]. BBs and MBs are mutually exclusive, i.e., a bug is either BB or MB. BBs manifest themselves consistently under certain types of conditions [24]. They are expected to be removed during the testing or debugging phases.

In this research, we adopt the BB and MB taxonomies for security defects to analyse vulnerabilities from the reproducibility perspective. Throughout the thesis, the security vulnerabilities with BB and MB characteristics are called BohrVulnerability (BV) and MandelVulnerability (MV), respectively. We assume that vulnerabilities can be classified into two categories: MVs and BVs. Due to the elusive behaviour of MVs, the usual testing techniques cannot always discover them. For instance, a detected vulnerability in Facebook in 2013 allowed anyone to write on other user walls. A user reported the vulnerability to Facebook engineers. However, it was ignored due to lack of sufficient information for reproducing the security failure [25]. The BV and MV definitions are completed in Chapter 3 and Chapter 4.

2.3 Overview of Machine Learning Techniques

Vulnerability classifications and prediction models have two main aspects: measurable attributes (called software metrics) and machine learning algorithms. The classification and prediction frameworks use different features that are described in (Chapters 3 and 4, respectively. As the relationships and dependencies between features and the target categories are not obvious, statistical and data mining techniques are
employed. Different classification algorithms are used as each technique considers various assumptions about the data under analysis which can influence the performance of vulnerability prediction. We select four widely used classification techniques. In the following subsections, they are reviewed.

2.3.1 C4.5 Decision Tree

Quinlan's C4.5 [26] is a popular Decision Tree (DT) algorithm inferred from the divide-and-conquer technique. It constructs an abstract decision tree by applying questions over attributes of instances to classify them. (e.g., Is number of developers higher than 10?). One attribute is selected as a root node. Then, it is divided into some child nodes. This procedure is applied to sub nodes recursively until no more division is possible. In the tree, root and inner nodes express decisions defined according to independent variables (features) and leaves represent the dependent variable (category). To identify the category of a vulnerability, the branches of the tree (the path from the root to the leaf nodes) are traversed based on the value of its feature vector.

2.3.2 Random Forests

Random Forests (RF) [27] is the aggregation of decision trees which are used for classification. Each tree uses different subsets of dataset, and use classification techniques according to the following steps: If there are $N$ samples, the algorithm picks randomly $n$ sub-samples for building the tree. Then, $m$ features out of $M$ are selected randomly to make sure that the trees are not highly correlated. One parameter of Random Forests is the size of $m$ which can be set by users. After constructing a set
of trees, a new sample enters to the forest for classification. Then, each tree votes on
the category of the instance. Finally, the category with the highest vote is selected.

Random Forest is a widely used technique in predicting the quality of software
systems due to its following advantages:

1. Typically Random Forest generates more accurate predictors than simple ap-
   proaches like Decision Tree or even advanced approaches [28].
2. It is more robust with respect to noise compared to other methods. In fact, it
   is effective in removing noise from data. This is considered a great benefit for our
   case study because data either in textual vulnerability report or code fix repositories
   contain noise.
3. In comparison with other classification techniques, Random Forest keeps the ac-
   curacy of prediction high even if correlated attributes exist. It is very helpful in
   this research because in our case study, the correlations between some attributes are
   undeniable. It impacts on the accuracy of rudimentary decision trees.
4. It works on a large data set effectively.
5. If a large number of data is missed, it is able to predict and keep the accuracy high.
6. It is able to estimate important attributes in classification.

2.3.3 Logistic Regression

Logistic Regression (LR) is a suitable statistical approach for finding the relationship
between dependent and independent variables [29]. In most cases, it is used for binary
classification. It calculates the probability of an instance belonging to the target
category. If the probability is bigger than a cut-off point (e.g., 0.5), the instance
is identified as the target class. The probability is computed based on following equation:

\[
P(Y = 1) = \frac{e^{\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n + \varepsilon}}{1 + e^{\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n + \varepsilon}}
\] (2.1)

In Equation 2.1, \(X_1, \ldots, X_n\), are independent variables. In fact, each \(X\) corresponds to a feature in the feature vector, and \(Y\) is a boolean output (0 or 1). The coefficients \(\alpha_1, \ldots, \alpha_n\) are parameters of the function that find the linear combinations of the explanatory variables. \(\alpha_0\) is a regression constant and \(\varepsilon\) is the probability of error. In our research, the algorithm predicts the likelihood of a vulnerability belonging to each category. If the likelihood of a vulnerability is bigger than a specified number (e.g., 0.5), it is classified as MV, otherwise it is classified as BV.

### 2.3.4 Naive Bayes

Naive Bayes (NB) [30] is a probabilistic supervised learning algorithm based on Bayes’ theorem. This simple algorithm calculates the posterior probability of a hypothesis. In the context of the classification framework, the hypothesis is that if a vulnerability is related to the BV or MV classes, \(C = BV, MV\). The feature vector \(x_1\) through \(x_n\) contains all collected and analysed information for the classification purposes. The following relationship states Bayes’ theorem.

\[
p(c_i|X) = P(c_i) \prod_{i=1}^{p} P(X_i|c_i)
\] (2.2)

The above formula is computed for each class (i.e., in the data mining field, the class, and category are identical) in \(C\). Finally, a class with the higher probability is
selected. The NB classifier assumes that attributes are independent of each other to simplify the approach. However, in reality, it might not be totally correct. There is some dependency between features. The NB algorithm gets accurate results with an arbitrary number of independent attributes even though it does not perform effectively for systems in which attributes are related.

2.4 Related Work

In this section, first available vulnerability classification techniques are discussed. As far as we know, there is no effort on security defect classification based on triggering conditions. We review studies for the defect classification to get an idea about vulnerability classification. Then, related work on software vulnerability prediction is described.

2.4.1 Software Vulnerability Classification

Different vulnerability classification techniques for different purposes such as devising verification and validation strategies, guidelines, and vulnerability detection tools are developed. Aslam et al. [31] propose a complete security defect classification scheme for the Unix operating system to help develop a security vulnerability detection tool. According to this research, the reason for synchronization errors is inappropriate evaluation of conditions. This view might not be true in all circumstances (e.g., fixing a security problem without any condition change).

Bishop [32] defines a vulnerability taxonomy with 6 dimensions (time of introduction, nature of the flaw, effect domain, source of identification, minimum number of
2.4. RELATED WORK

components to exploit a vulnerability, and exploitation domain) in the Unix operating system. These dimensions cannot separate security defects based on software functionality. Krsul [33] argues that most of the classification schemes are ambiguous. More importantly, the rapid growth of new undiscovered vulnerabilities reduces the effectiveness, and accuracy of the previous manual vulnerability classification approaches. In the following, we review defect classification techniques based on the BB and MB concepts.

2.4.2 BB and MB Classification

There are some empirical studies on large industrial software systems based on the reproducibility of software failures. Grottke et al. [21] study 18 JPL/NASA space missions to find out available BBs and MBs in the projects. Their results show that even if well-known testing techniques are applied to the system, a large number of BBs stay in it. Furthermore, the Tandem system [18] shows that a significant proportion of bugs in large and complicated systems are related to MBs. These empirical studies achieve interesting results about the distribution of BBs and MBs in big systems, but security issues are not seriously analysed.

Cotroneo et al. [34] develop a BB and MB classification approach based on bug reports. They conclude that MBs need more time to fix, and special strategies should be employed to address them. Xia et al. [35] propose a text mining-based solution named USES to classify bugs into BBs and MBs. The approach is applied to textual bug reports.

The two mentioned approaches (the BB and MB classification and USES) develop security vulnerability classification models solely according to textual bug reports.
2.4. RELATED WORK

These approaches cannot provide accurate results due to possible noises in texts. For instance, ’error’ and ’test’ terms might not be good indicators to distinguish BBs from MBs. Because these terms can be used to describe any defect types. Performance of classification approaches based on bug report highly depend on the effectiveness of the term selection and noise removal.

Another weakness of the aforementioned approaches is that they do not distinguish between different types of defects such as performance and security. However, security defects require more attention due to their potentially severe consequences. The focus of this research is on security vulnerabilities because they need to be handled differently. Moreover, models combining different types of features related to bug reports and code fixes perform relatively better than the models that are built based on bug reports or code changes [36]. In our classification framework, both vulnerability reports and code fixes are used.

2.4.3 Software Vulnerability Prediction

A vulnerability prediction model identifies program units (e.g., modules, files, and functions) which are more sensitive to attacks. It helps security experts to increase their inspection and testing on the detected parts in order to mitigate and eliminate a large number of security breaches with less inspection and effort [37].

In the last decade, a small number of studies have focused on vulnerabilities because of the existing challenges (presented in the next section). However, a large number of papers in the field of the defect prediction are published. The defect prediction techniques cannot be directly used for the vulnerability prediction due to the small number of detected security issues.
2.4. RELATED WORK

The previous literature shows that the performance of a vulnerability prediction model changes with respect to metrics. Different software metrics are defined and employed for vulnerability prediction. Zimmermann et al. [9] investigate the relationship between vulnerabilities and the classical software metrics (e.g., complexity, churn, coverage, dependency measures, and organizational structure). They then use Logistic Regression to predict vulnerabilities based on the metrics. Their experimental analysis on Windows Vista shows that the approach can predict vulnerabilities with high precision (60%) and low recall (40%).

Shin et al. [10] explore the correlation between vulnerability and complexity, code churn, and developer activity. By adopting the linear discriminative analysis and Bayesian Network, it has been shown that the metrics are really predictive, and a complicated code is more vulnerable. Chowdhury et al. [12] investigate the relationship between CCC metrics (complexity, cohesion, and coupling) and security defects. They then build a vulnerability prediction model to anticipate vulnerabilities at the code level. Gegick et al. [38] predict security defects by using security-related static analysis alerts [39], code churn, size, and inspection faults. The model predicts that 75.6% of vulnerabilities of the Cisco software are related to 18.6% of the vulnerable components.

Some researchers consider source code as text to apply text mining-based techniques on source code instead of using classical software metrics. Hovsepyan et al. [11] suggest an accurate text-based vulnerability prediction model which analyzes the source code. Furthermore, Scandariato et al. [37] propose a text mining-based feature selection. Files are represented by frequency vectors. Then, features and machine learning techniques are used for predicting the location of vulnerabilities. Neuhaus et
al. [8] study the relationship between imports, functions, and vulnerabilities for the Mozilla project. They develop a machine learning-based tool to predict the vulnerable components. It achieves the precision of 70% and recall of 45%.

All the above mentioned approaches use the classical software metrics to find the location of vulnerabilities. Although the approaches provide information about vulnerability locations and characteristics, their precision and recall are not high enough to be used in practical applications. Moreover, any of them do not study vulnerabilities from the reproducibility perspective.

### 2.5 Challenges in Vulnerability Study

There are some challenges in studying vulnerabilities. First, usually, researchers and developers are unwilling to expose information about security problems to decrease the chance of future attacks. Accordingly, vulnerability reports do not give adequate information to realize where (version or location) the security failures happen. Second, vulnerabilities occur rarely, and they are sparsely distributed. The scarcity of vulnerabilities in software makes statistical analysis difficult. Third, all security faults cannot be identified by failure testing [40]. Due to these challenges, vulnerability data has lower quality than defect data in software repositories. Hence the vulnerability classification and prediction field is not mature enough. According to Zimmermann et al. [9], dealing with vulnerabilities is similar to "searching a needle in a haystack".

### 2.6 Summary

This chapter starts with basic terminologies that are required to understand the proposed frameworks. Although studying security vulnerabilities is important to
improve the security of large projects, there are only few research initiations in this direction. Therefore, we first analyse the defect classification approaches based on the reproducibility concept. We classify vulnerabilities into two groups: easily and hard to reproduce security failures.

An important part of the vulnerability classification and prediction frameworks is machine learning techniques. An overview of the well-known machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are provided.

Then, the related work on vulnerability classification approaches is summarized. As there is no effort on classifying vulnerabilities based on the reproducibility of security failures, the related work on defect classification based on this concept is presented. For the vulnerability prediction, most of the approaches use the classical software metrics to identify the location of vulnerabilities in the source code. Their results are compared and contrasted to indicate the effectiveness of the software metrics for the vulnerability prediction. Then, existing challenges in vulnerability study are stated.
Chapter 3

A Vulnerability Classification Framework

Vulnerability classification is a fundamental approach for visualizing and analysing information about existing vulnerabilities in a system. This chapter presents a vulnerability classification framework based on the reproducibility of security failures. It is able to differentiate between BVs and MVs based on the textual reports and code fixes. The components of the framework are described.

Then, this chapter reviews Mozilla Firefox. It is selected as our case study to empirically validate the effectiveness of the classification framework. The procedures of the data collections, extractions, and analyses are described. Then, we balance the data set to avoid the over-fitting problem. To employ machine learning techniques, the parameter tuning for the techniques is presented. Then, they are applied to the data set. The results are evaluated based on the well-known evaluation metrics. The experimental evaluations and results are described.

3.1 Classification Framework Overview

An overview of the proposed classification framework is illustrated in Figure 3.1. It has two phases: training and deployment. The training phase associates a set of
3.1. CLASSIFICATION FRAMEWORK OVERVIEW

vulnerabilities with vulnerability reports, code fixes, and the known vulnerability categories (see Section 3.3). After extracting the defined features, the most discriminative features are identified. Then, a model which can distinguish BVs from MVs is constructed. The deployment phase has two inputs: the model from the training phase and a new vulnerability with an unknown class. Then, the framework predicts if the vulnerability can be classified into BV or MV. The phases and components are described in the next subsections in more detail.

3.1.1 Training Phase

The training phase has five components: Textual Feature Extractor, File Extractor, Code Feature Extractor, Discriminative Feature Extractor, and Model Builder as Figure 3.1 shows. In this framework, two types of features are defined: features related to textual vulnerability reports and code fixes. The main goal of the feature definition is to represent vulnerabilities with the quantitative aspects. The Textual Feature Extractor component extracts feature values by applying heuristics on the title, description,
and summary parts of textual reports. The relevant changed files are extracted by the File Extractor component. Then, the code level features are computed by the Code Feature Extractor component. As all the defined features might not be suitable to distinguish BVs from MVs, the Discriminative Feature Extractor component is defined to find the most relevant features. Each vulnerability is represented by a feature vector. Different well-known machine learning algorithms (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are used to find the relationship between the defined features and the BV or MV characteristics. A classifier with the highest MV F-measure is selected and is passed to the next phase.

### 3.1.2 Deployment Phase

The deployment phase has four components: Textual Feature Extractor, File Extractor, Code Feature Extractor, and Category Predictor. The first three components are similar to the training phase. The deployment phase extracts the value of the most relevant features. In the next step, the Category Predictor component takes the model from the training phase to identify the category of an unseen vulnerability. More information about the framework components can be found in the next sections.

### 3.2 Framework Components

In this section, the components of the classification framework are presented in detail.

#### 3.2.1 Textual Feature Extractor

In most of the issue trackers, the conditions that activate vulnerabilities (called activation conditions) are not explicitly mentioned. We first pre-process textual reports
of a bug tracker (e.g., Bugzilla) to identify any extractable information. Also, previous successful software defect and vulnerability classification approaches are reviewed to get hints. The features which may have direct or indirect relationships with vulnerability reproducibility are analysed.

Some of the utilized metrics in this research have been successfully used for classification. For instance, Cotroneo et al. [34] analyse bug reports of four large OSS (Open Source Software) projects (Linux, MySQL, HTTPD, and AXIS) with respect to time to fix and severity. Furthermore, Landwehr's Taxonomy [41] gives us the understanding that locations of security flaws, which refer to as places where the security vulnerabilities are manifested, can be dimensions for the vulnerability categorization.

Some information is not documented in the bug repositories such as activities (done during the defect discovery) and target attributes (containing the defects). Moreover, the impacts of security defects (on confidentiality, integrity, and availability) are not mentioned explicitly, and they require significant manual effort to identify the impacts. From the reports, we cannot also understand when and how a security flaw enters the system. Hence, we ignore these features.

Overall, five main dimensions are selected: time to fix (TTFX), developer, the number of vulnerabilities (NOV), location, and importance. We assume that these dimensions can be important in our analyses. We mine different parts of the vulnerability reports (including title, description, and summary) to extract features by using Python script and C code. The features with their corresponding extraction mechanisms are described in the following subsections.
3.2. FRAMEWORK COMPONENTS

Time to Fix (TTFX)

TTFX is a time period that developers spend to fix a security problem. It can show the impacts of vulnerabilities on the security defect management. Cotroneo et al.'s [34] analyses disclose that the type of a defect noticeably influences the TTFX in large systems. We hypothesize that since MVs are more complicated than BVs in terms of reproducing security failures, they may need longer TTFX than BVs.

**Extraction mechanism:** In the vulnerability repositories, when a security failure is detected, the time is recorded as reported date (RD). When it is fixed, the time is reported as modified date (MD). During the time between RD and MD, the vulnerability is open. The time period includes four steps:

*Step 1.* Developers reproduce the security failure.

*Step 2.* Developers try to identify its root cause. A security expert reviews the results of testing tools to decide if the vulnerability is exploitable.

*Step 3.* If the vulnerability needs to be fixed, it will be assigned to an appropriate person to implement the suggested solution, and this time period is called triage time. Assigning the security defect to a right developer might take a long time.

*Step 4.* Finally, the proposed approach is validated through testing.

Although the expected TTFX for MVs is higher than BVs, it should not prejudge our comparison. We make an assumption that sometimes TTFX of a vulnerability might be influenced by other factors. For instance, involved developers might be busy with a large number of vulnerabilities simultaneously. Hence, the next feature considers the role of developers.
3.2. FRAMEWORK COMPONENTS

Developer

This feature demonstrates the correlation between developers and vulnerability types. In our case, developers refer to security response teams, developers, and security experts. It has two subsets.

- Number of Developers (NOD): NOD measures how many security experts are required to fix a vulnerability. It is the number of unique developers involved in the vulnerability correction process. We assume that vulnerabilities with complicated activation conditions engage more developers.

- Developer Experience (DE): We hypothesize that for security issues which are harder to reproduce, the knowledge of security testers or experts can be effective to deal with similar issues. Developers who have experienced a large number of MVs might be more expert than others to address complicated vulnerabilities.

**Extraction mechanism:** To identify NOD, the number of unique developers who write comments, and try to address a security issue is counted. Sometimes, a developer may write more than one comment. However, we only count it once. Also, if a developer uses an email address instead of its name, the name part of its email is considered. To extract DE, we assume that each vulnerability has two types of developers: assigned developer (AD) and other developers. We extract the AD for each vulnerability. When ADs of all vulnerabilities are identified, we check the data set to find how many vulnerabilities have been fixed by each AD.
3.2. FRAMEWORK COMPONENTS

Number of Vulnerability (NOV)

When a vulnerability is under analysis, other security intrusions might be introduced into the system that can influence the fix process. This feature considers the number of other vulnerabilities. We analyse the AD by defining two following features.

- Number of Vulnerabilities with an identical Assigned Developer (NOV-AD): We assume that a vulnerability might not need a long time to fix. However, a list of other vulnerabilities waiting to be processed by the same AD extends that time.

- Number of Vulnerability (NOV): This feature counts other vulnerabilities in the data set without considering if they have similar AD or not.

Extraction mechanism: We analyse all the RD and MD of existing vulnerabilities in the data set. Then, vulnerabilities detected during the open time of the vulnerability under analysis are considered as influential vulnerabilities.

Location

The location indicates the place where a vulnerability occurs. The location can be one of the important factors for the vulnerability classification. We consider components, risky files, and versions.

- Component: The attribute finds failed components/subsystems in products.

- Risky files: During the fix process of security vulnerabilities, some files are changed. These files are considered as risky files in this thesis.
3.2. FRAMEWORK COMPONENTS

- Version: The version feature can show the distribution of different vulnerability types in different versions. Large corporations release different versions frequently to improve the quality of software. However, some vulnerabilities remain in them for a long time.

**Extraction mechanism:** In the most vulnerability repositories, components and versions in which security breaches happen are explicitly documented. Vulnerable file extractions from textual reports require a considerable amount of manual effort. To automatically extract the risky files, we use a revision control system. The detail of extraction mechanisms for vulnerable files is described in Section 3.2.2.

**Importance**

The motivation behind using the "importance" feature is to realize if end users perceive different behaviour from security failures caused by BVs and MVs. We assume that the "importance" can have two subcategories:

- **Severity:** The severity becomes relevant when a vulnerability is exploited- that is, it measures how severe its consequence will be from users’ perspective. It can have different levels. Table 3.1 shows an example of different severity levels for Mozilla Firefox in the Bugzilla repository. A vulnerability within the critical severity level can have a serious impact on a system (e.g., a crash, loss of data and a severe memory leak).

- **Priority:** It indicates which vulnerability deserves higher priority to be addressed in comparison with others.

**Extraction mechanism:** Developers report the severity of each security failure from
3.2. FRAMEWORK COMPONENTS

their own perspectives. This feature value can be extracted directly from the repository fields of the vulnerability database. Priority can help engineers and programmers prioritize the vulnerability fix. Sometimes, this field is not set for all vulnerabilities. Only the severity feature is considered.

Table 3.1: Example of different levels of severity in Bugzilla [1]

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocker</td>
<td>Blocks development and/or testing work.</td>
</tr>
<tr>
<td>Critical</td>
<td>Crashes, loss of data, severe memory leak.</td>
</tr>
<tr>
<td>Major</td>
<td>Major loss of function.</td>
</tr>
<tr>
<td>Normal</td>
<td>Regular issue, some loss of functionality under specific circumstances.</td>
</tr>
<tr>
<td>Minor</td>
<td>Minor loss of function, or other problem where easy workaround is present.</td>
</tr>
<tr>
<td>Trivial</td>
<td>Cosmetic problem like misspelled words or misaligned text.</td>
</tr>
<tr>
<td>Enhancement</td>
<td>Request for enhancement.</td>
</tr>
</tbody>
</table>

3.2.2 File Extractor

The motivation behind finding the risky files is based on a hypothesis that in software systems most of the faults belong to a small number of modules [42–44]. Identifying parts of a system that are potentially prone to vulnerabilities, is really helpful for testers and quality managers. It can guide a software development community to increase testing efforts on parts of the system that are prone to vulnerabilities to improve the security of software in next versions. It can also help meet the time and budget limitations.

Bug trackers store security failures that are reported by users or developers. More information about bug fixes from the code perspective (e.g., how, when, and who fixed the security issues) are available in source control systems (e.g., Mercurial,
CVS (Concurrent Version System)). They enable developers to track changes in the source code. Unfortunately, not all changes fix problems, but also they can add more problems in the system. In revision control systems, developers log all modified files along with additional information (e.g., id, time, date) when they fix an issue. On the other hand, the code repositories do not provide information about reasons for changes such as fixing a bug or introducing a feature into the system. Both bug tracker system and source code management are required. However, they are not directly linked.

We use Mercurial\(^1\) (known as 'hg'), a distributed revision control system to extract files related to the security failures in the data set. After installing Mercurial, we checkout/clone to get a local copy of the repository (database of files and historical data). We use the "hg log -v" (the "-v" argument lists all files). The "hg"\(^2\) command provides a command line interface to the Mercurial system to download commit logs of the project. Figure 3.2 indicates an example of a log message. To map the vulnerabilities to the modified files during its fix process, we employ the approach proposed by Neuhaus et al. [8] and Sliwerski et al. [45]. If a vulnerability id is found, relevant information (e.g., date, time, summary, revision number and parent revision number, etc) is stored for more analysis (see Chapter 4).

There might be more than one commit for each vulnerabilities. We collect all of the corresponding information. This integration technique depends on the comments of developers. In the revision control systems, many changes are made to enhance a part of systems without addressing any security issue. We focus on C/C++ files and their header files. Other file types such as scripts, configuration, and make files are

\(^{1}\)https://www.mercurial-scm.org/
\(^{2}\)https://hg.mozilla.org/mozilla-central/help
There is an alternative approach to find the risky files. Files with vulnerabilities can be extracted from the HTML files by running Python scripts that parse the files. For instance, in the Bugzilla repository, developers provide a link (that is named *diff*) to the changed files.

### 3.2.3 Code Feature Extractor

The Code Feature Extractor component is responsible for code related metrics. It is defined to understand the correlation between changes and fixes in the source code and the vulnerability types. We conjecture that the complexity of code fixes can be a reason for the MV complexity. To measure the quantitative complexity of the vulnerability code fix, three features based on the risky files are extracted: Number of Changed LOC (NOC-LOC), Number of Changed Files (NOC-F), and Fix Entropy [46]. They are described in the following subsections with their corresponding extraction mechanisms.
3.2. FRAMEWORK COMPONENTS

The Number of Changed (added/deleted) Line Of Code (NOC-LOC)

We assume that more changed Lines of Code to fix a security issue are associated with more code complexity. Files with a large number of changes are more likely to be vulnerable.

The Number of Changed Files (NOC-F)

The Number of Changed Files means how many files have been changed during the fix process of a vulnerability. It is believed that the fix process of a complicated code requires a large number of changes [47].

Vulnerability Fix Entropy

To fix a security issue, several files might need changes (e.g., addition and deletion). It would not be easy for developers to follow a fix process with many modifications [47]. We assume that one reason for MV complexity can be related to the complexity of the fix process. Fixing complicated vulnerabilities (MVs) can involve many files in comparison to simple vulnerabilities (BVs). Consequently, they are expected to have high entropy. We compute the entropy of vulnerability fix by applying Shannon's entropy [48], defined in Equation 3.1.

\[ H_n(v) = -\sum_{i=1}^{n} p_i \times \log_n(p_i) \] (3.1)

Where \( v \) is a vulnerability, \( n \) is the number of files that have been changed during the vulnerability fix, \( p(i) \) is the probability occurrence of the specific file \( i \) which is changed to fix vulnerability \( v \). Entropy is maximum (\( H_n(v) = 1 \)) when all files have the same probability value \( p_i = \frac{1}{n} \forall i \in (1, 2, 3, ..., n) \), which means that a large
number of files have been changed. On the other hand, the entropy will be minimum \( H_n(v) = 0 \) when only the vulnerability is concentrated in one file. Here is an example of how the entropy of a vulnerability is computed. A problem is fixed by changing two files: \( file_A \) and \( file_B \). In \( file_A \): 2 and 5 lines of code are added and deleted, respectively; in total, there are 7 changes. In \( file_B \): 4 and 1 lines of code are added and deleted, respectively; there are 5 changes. To compute the probability of each file, we divide its changed LOC by all changed lines (12 in this case). We have \( p(file_A) = \frac{7}{12}, p(file_B) = \frac{5}{12} \). To normalize Shannon's entropy, Equation 3.1 is divided by \( \ln(n) \). For this example, the normalized Shannon's entropy is 0.980864. If all changes are scattered over 12 files instead of 2 files, the entropy of vulnerability fix becomes 1. Vulnerability fix with 0.980864 entropy is more concentrated and less complicated.

**Extraction mechanism:** The NOC-LOC and NOC-F are straightforward features. For each vulnerability in the data set, the NOC-LOC and NOC-F are computed for the relevant vulnerable files (based on its revision number and its parent(s)' revision numbers extracted by *File Extractor* component described in 3.2.2). We then run
3.2. FRAMEWORK COMPONENTS

hg diff -stat -r #parent_rev -r #vul_rev command for each file. Figure 3.3 shows an example of running the command for a vulnerability with the revision number 267263. Sign (+) refers to insertion and sign (-) refers to deletion. Furthermore, the number of changed files is another output of this command. After extracting all reported files for the vulnerability, its fix entropy is computed based on Equation 3.1. We implement the extraction mechanism of the code related features in Java.

3.2.4 Discriminative Feature Extractor

The features are used according to our hypotheses and previous work achievements. All the dimensions may not have enough capability to differentiate between the BV and MV categories. This is because irrelevant or redundant features can decrease the efficiency and performance of the classifiers. To resolve this issue, Fisher score (or Fisher kernel) [49] is successfully used to identify the most relevant features. The technique calculates scores based on the standard deviation and means of features. We exploit discriminative dimensions by calculating the Fisher score for each feature individually based on Equation 3.2 mentioned in [50].

\[
F(j) = \frac{\sum_{c=1}^{\#class} (\bar{x}_c^j - \bar{x}_j)^2}{\sum_{c=1}^{\#class} (s_j^c)^2}
\]

Where \(F(j)\) shows the Fisher score of feature \(j\), \(\bar{x}\) and \(\bar{x}_j^c\) denote averages of the feature \(j\) for the whole data set and \(c\)-th class, respectively. \(s_j^c\) is the variation for feature \(j\) in the \(c\)th class. The \(n_c\) is the number of data instances in the \(c\)-th class. Fisher score is a value between 0 and 1. A feature with a score near 1 is the most discriminative feature, while a feature with a score near 0 is less discriminative.
3.2.5 Model Builder

After representing vulnerabilities with the most discriminative features, we need to know their relationships with the type of vulnerabilities (BV vs. MV). The relationship might change for other systems. The goal of the Model Builder component is to build a classifier which can learn characteristics of each class based on the feature vectors. This technique is widely adopted to address knowledge discovery problems.

A set of inputs with their corresponding categories (BV vs. MV) is given to the system to train the model. Then, the model learns the BV and MV characteristics according to the features of the instances in the data set. Various machine learning techniques are available and can be used to build models. Each technique makes different assumptions about input data which can affect the performance of the classification. Different techniques may have different performance. However, the differences are not significant [51]. The most well-known machine learning techniques (reviewed in Chapter 2) are selected for building models.

3.2.6 Category Predictor

The Category Predictor component takes the classifier to identify the category of an unseen vulnerability. It calculates how likely it is that a vulnerability belongs to the BV or MV categories. Finally, a category with the highest probability is selected.

3.3 Case Study Design

We select Mozilla Firefox as our case study for the following reasons. The project is one of the most popular open source internet browsers. It is large in terms of the
number of users (approximately 270 million). The Mozilla project is a large code-base OSS (Open Source Software). It is one of the largest OSS, and it is as big as the Linux kernel. In 2001, David Wheeler's study [52] showed that each release of Mozilla Firefox has more than 2 million lines of code. A huge number of developers are involved in the project. A part of developers are paid by Netscape, Sun, IBM, and a noticeable number of them are volunteers [52].

3.3.1 Bugzilla

Mozilla Firefox uses Bugzilla as a bug tracking system. More than 100 corporations such as KDE, GCC, Apache, and Eclipse manage their bug reports using open bug repositories. This repository documents 567,595 bugs from the Core, Firefox, Sea-Monkey and Thunderbird products. It reports all kinds of bugs such as security and performance. Each bug is represented by a unique id. The textual bug reports contain more attributes such as open date, modified date, assigned, component, severity, procedure of reproducing failure, and patches. A sample of a textual bug report is shown in Figure 3.4.

Most of the bug tracking systems provide a considerable amount of historical information about reported bugs. However, this raw information is not sufficient to avoid the occurrence of security failures in the next releases, it needs more analysis.

Mozilla Foundation Security Advisory (MFSA)

We use Mozilla Foundation Security Advisory (MFSA) [32] to extract security issues. Vulnerabilities in the bug tracker systems are recognized by an expert security
Figure 3.4: A textual report sample of Bugzilla
3.3. CASE STUDY DESIGN

advisory team. Hence, we do not need to calculate the precision and recall of vulnerability identification. Vulnerability reports in the Bugzilla system like many other bug tracking systems contains three main categories of information:

- **Description**: The misbehaviour of the software is described in natural language. Furthermore, possible locations in the source code which might be related to the vulnerabilities are stated.

- **Comments**: After contributors’ discussions about root causes of the security fault, a fix method is suggested. Then, it is validated through testing. Moreover, the related patch(es) are referenced in comments.

- **Test case**: A test case represents how to reproduce the security failure.

Vulnerability Collection Approach

In this study, vulnerabilities are collected from different versions to make sure that a specific version does not influence classifiers. The time difference between each Firefox release is less than two months. In addition, each release has few vulnerabilities [53], and it is not enough for vulnerability analysis. Based on previous software vulnerability analyses [31,33,54,55], most vulnerabilities fall into five main categories:

1. **Memory Fault (Mem)**: Faults in memory can lead to memory corruption, memory safety error, and memory leak. One main cause of memory fault is direct access to the memory through allocation and deallocation functions of C and C++.

2. **Null Pointer (NP)**: An NP can result in segmentation faults. When a pointer has NULL value instead of a valid memory address, and an application refers to it, NP exception occurs. If an attacker can use an NP, he or she is able to violate the security policy of the system resulting in the system crash or information disclosure.
3. Initialization-Validation-Checking (IVC): Using resources without any appropriate initialization may cause information leaks or segmentation faults. Three elements including input, origin, and target should be validated appropriately.

- **Input validation**: Numbers, formats, and types of all inputs (e.g., text, video, audio, and command line input) should be carefully checked. Attackers try to inject malformed data to the program in order to take the control of the system. An invalidated input may cause some problems including system crash, buffer overflow, code insertion, format string vulnerability, URL command, and social engineering.

- **Origin validation**: It makes sure that the origin is the same as it claimed.

- **Target validation**: Identity of the target should be checked to avoid sending information to an untrusted target.

4. Race Condition (RC): A race condition occurs when shared data can be manipulated and accessed by multiple processes simultaneously. If the correct behaviour of the system relies on a special order of execution, the system has a flaw. All kinds of errors related to time and serialization cause race conditions. When an attacker can take advantage of the vulnerability, he or she can change a filename, insert a malicious code, or gain unauthorized access to the system. For example, the UNIX operating system reads the file `/etc/passwd` and `/etc/shadow`. If an attacker can add information to both files, the attacker can create a user or even root user to the system.

5. Access Control (AC) (Authentication-Authorization-Cryptography): Actions allow to gain an appropriate permission to obtain the control of whole resources. The
system allows a protected operation to be invoked by an agent, while its authority is not validated. A system without strong authentication mechanisms may lose the control of some parts of the system. For instance, a system which is vulnerable to SQL injection, and cross-site scripting, authentication mechanism allows hackers to authenticate themselves as valid users (normal or administrative users). When an attacker logs in as an administrative user, the attacker can achieve the appropriate permission to control the whole resources.

An authorization vulnerability is dangerous because an attacker can easily exploit it and gain highest privilege level access. Attackers can remotely run commands on the device without needing the password, and gain the control of the system.

Some vulnerabilities in cryptography (e.g., the statistical definition of seeds, salts, initial vectors, and keys) enable attackers to get them by decompiling the app. The stateless encryption algorithm can help attackers decrypt data. Attackers utilize different methods to achieve sensitive information (e.g., transmitted data, stored data, and password on devices). With this vulnerability, data is easily decrypted, and used for transaction.

Figure 3.5 shows an overview of the proposed vulnerability collection. In this work, we identify vulnerabilities belonging to these five categories by using heuristics. We search for "memory", "validation", "initialization", "null pointer", "race condition", "authorization", "authentication", and "cryptography" keywords in MFSA links. Then, for each selected link, a page providing a brief Discretion and References is shown in MFSA. Figure 3.6 indicates an example. In the References section, we can link to each vulnerability report of Bugzilla.

3.3. CASE STUDY DESIGN

Figure 3.5: An overview of the vulnerability collection approach.

Figure 3.6: A vulnerability report in MFSA
3.3. CASE STUDY DESIGN

We store all links in the database. Each defect has a unique id which can be retrieved from URL address. For example, https://bugzilla.mozilla.org/show_bug.cgi?id=1225250, the last part of the URL, id=1225250 is the id of the vulnerability. Before sending vulnerability to the next step for the vulnerability type detection, they are filtered based on their status to have a reliable and unique data set. It is described in the following section in more detail.

BV and MV Category Identification

After collecting vulnerabilities, we conduct manual analysis on the different field of textual reports (e.g., title, description, and summary fields). We consider the reproducibility of security vulnerabilities and the definitions of BV and MV (see Chapter 2) rather than developers’ perspective which can contain errors to differentiate BVs from MVs [34]. We manually classify vulnerabilities by following five steps:

Step 1. Remove vulnerabilities the fixing of which would not improve the quality of the software (e.g., duplicate and enhancement). A large number of vulnerability reports are duplicated. We search in the textual reports to understand if they are caused by another vulnerability which exists in our data set to remove duplicates. Invalid, rejected, unconfirmed vulnerability reports, and vulnerability reports for enhancement are not considered in our analysis.

Step 2. Understand vulnerability activation. We search vulnerability reports to understand conditions which activate a security vulnerability (data inputs and events) and how the security failure affects system states.

Step 3. Identify MVs. A vulnerability is considered as MV when the information in the textual report shows complicated and transient behaviour from vulnerability.
3.3. CASE STUDY DESIGN

For instance, the vulnerability occurrence depends on system internal environment (hardware and operating system), the timing of input or sequence of operation. Also, the time delay between vulnerability activation and error occurrence, concurrency, resource leak, and a wrong memory state are symptoms of MVs. In textual reports of security issues, some resolution terminologies denote hard to reproduce security failures including "works for me", "cannot reproduce", and "work on my machine".

*Step 4. Identify BVs.* If the vulnerability is not recognized as MV, and it is always reproducible, it is classified as BVs.

*Step 5. Exclude reports with lack of information.* Some textual reports lack enough details for classification. We do not consider these vulnerabilities in our analysis.

Table 3.2 contains two examples of BV and MV. The vulnerability with id 387333 is always reproducible; therefore, it is classified as BV. In contrast, the security issue with id 486269 is MV because it cannot reproduced under any circumstance.

Table 3.2: Example of BV and MV

<table>
<thead>
<tr>
<th>Vul ID</th>
<th>Title</th>
<th>Description</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>387333</td>
<td>Unauthorized access to wyciwyg:// documents possible.</td>
<td>Access wyciwyg:// documents without proper same domain policy checks through the use of HTTP 302 redirects. &quot;It is always reproducible&quot;.</td>
<td>BV</td>
</tr>
<tr>
<td>486269</td>
<td>Race condition [while accessing the private data of an NObject JS wrapper class object].</td>
<td>The vulnerability is caused due to a race condition in xul.dll, while accessing the private data of an NObject JS wrapper class object when navigating away from a web page during loading of a Java applet. &quot;I’ve only been able to reproduce this crash once myself&quot;.</td>
<td>MV</td>
</tr>
</tbody>
</table>

4https://bugzilla.mozilla.org/page.cgi?id=fields.html
3.3. CASE STUDY DESIGN

Vulnerability Analysis

We have collected software vulnerabilities from MFSA related to five discussed categories from January 21, 2005 to January 13, 2015. Vulnerabilities with CLOSED status (containing RESOLVED\(^{5}\) and VERIFIED\(^{6}\)) are selected to make sure that appropriate attempts have been taken by security experts to address issues. These vulnerabilities can provide information about the fix process and reproducibility of security failures. In total, 725 security vulnerabilities from thirty-nine releases of Mozilla Firefox from R1.0 (2004-11-09) to R39.0 (2015-07-02) are collected.

In spite of our assumption that BVs can be detected and eliminated during the testing phase, we realize that Mozilla Firefox has a non-negligible number of post-release vulnerabilities related to BVs (386 out of 725). One possible reason is that practical testing of large systems is difficult or there is no adequate testing technique. In addition, even if we assume that BVs are detected and fixed in the earlier stages, expanding or modifying the project can introduce new vulnerabilities to the system. In our case study, the distribution of BVs and MVs are almost similar. 53% and 47% of vulnerabilities belong to BVs and MVs, respectively as shown in Table 3.3. The results show that there is a direct correlation between the project size and the number of security issues.

Table 3.4 indicates the absolute numbers and percentage of the BV and MV distributions for each security defect type. The predominant proportion of detected vulnerabilities (702 out of 725) belongs to the memory. More interestingly, the proportion of BVs and MVs in the memory category is similar too. The memory related

\(^{5}\)A defect with RESOLVED status can be reopened if the solution is not correct. If the resolution is fixed, it is changed to VERIFIED.

\(^{6}\)A defect with VERIFIED status waits for QA verification. If the verification is confirmed by QA, it is closed.
security faults increase the security failure rate and decrease the performance.

Table 3.3: Distribution of BVs and MVs

<table>
<thead>
<tr>
<th>#BV</th>
<th>%BV</th>
<th>#MV</th>
<th>%MV</th>
</tr>
</thead>
<tbody>
<tr>
<td>386</td>
<td>0.53</td>
<td>339</td>
<td>0.47</td>
</tr>
</tbody>
</table>

Table 3.4: Distribution of BVs and MVs for different categories in the data set

<table>
<thead>
<tr>
<th>Vul Type</th>
<th>#BV</th>
<th>%BV</th>
<th>#MV</th>
<th>%MV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>373</td>
<td>53</td>
<td>329</td>
<td>47</td>
</tr>
<tr>
<td>NP</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>IVC</td>
<td>4</td>
<td>0.44</td>
<td>5</td>
<td>0.55</td>
</tr>
<tr>
<td>RC</td>
<td>1</td>
<td>0.25</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>AC</td>
<td>7</td>
<td>0.875</td>
<td>1</td>
<td>0.125</td>
</tr>
<tr>
<td>Total</td>
<td>386</td>
<td>53</td>
<td>339</td>
<td>47</td>
</tr>
</tbody>
</table>

The frequency of each category gives hints about the type of testing. A system with a large number of MVs requires additional testing and recovery strategies. However, functional tests are recommended for a system with large number of BVs.

The BV and MV distributions depend on the nature of systems under analysis. For instance, how often a system interacts with hardware devices or with which language it is developed. The results can be reliable for systems developed in C/C++. Since the memory is managed by developers, it is more prone to memory related vulnerabilities. In Java, the memory is automatically managed by the garbage collector. The advantage of our approach is that it is independent of the implementation details. Although the analysis is conducted for one project, it can be easily expanded for other large open source systems.

Then the location of 725 security vulnerabilities in the source code is extracted based on the approach presented by the File Extractor component. In this research, the location of a security vulnerability indicates the location of the vulnerability.
3.4 Experimental Evaluation and Results

In this section, we make the data set balanced and identify the most correlated metrics to make it ready for the experiment. The parameter tuning for machine learning algorithms (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are presented. Then, we review the widely used evaluation metrics to assess the effectiveness of the classifiers. Finally, the classifier results are discussed.

3.4.1 Data set Setup

In our data set, 53% of instances belong to the BV categories and the rest of the vulnerabilities are of the MV category. Even through the data set is not highly imbalanced, we make it balanced to avoid bias. Applying the machine learning techniques on imbalanced data set can increase the likelihood of over-fitting [56] occurrence. It means that the category with dominant instances biases the classifiers toward itself. In this case, a model with high accuracy is not trustworthy.

Different approaches can be employed to make the data set balanced. First, the majority category is under-sampled. Under-sampling is widely used [57] for large data sets because losing a part of data does not seriously impact the result. However, this technique is not suitable for a data set with a small number of instances (e.g., our data set) since a considerable proportion of data from which information can be extracted is wasted.

Second, the class with minority instances is over-sampled [58, 59]. It is used for
data sets with a small number of instances. It creates samples for a category with a small number of instances in the data set. We use SMOTE (Synthetic Minority Over-sampling Technique) algorithm [58]. It over-samples the minority class by generating synthetic instances instead of replacing them. In the minority category, it randomly selects $k$ nearest neighbours and joins them. In our experiment, $k$ is set at 5.

After making the data set balanced, each vulnerability is represented by a feature vector. Then the *Discriminative Feature Extractor* component (as described in Section 3.2.4) calculates Fisher scores for all features. The top-ranked relevant features with large Fisher scores are shown in Table 3.5. The severity feature is not a discriminative feature because of its low score. Hence we can conclude that although MVs show more complicated behaviour than BVs, the perceived security failures caused by MVs are not significantly different from the security failures due to BVs from users’ or developers’ perspective. It is consistent with Cotroneo et al. [34] in finding defects. The hypotheses about the developer experience (DE) and location (versions and components) are rejected as well.

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Fisher Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix Entropy</td>
<td>0.13</td>
</tr>
<tr>
<td>NOD</td>
<td>0.104</td>
</tr>
<tr>
<td>NOC-CF</td>
<td>0.08</td>
</tr>
<tr>
<td>NOV-AD</td>
<td>0.04</td>
</tr>
<tr>
<td>TTFX</td>
<td>0.034</td>
</tr>
<tr>
<td>NOC-LOC</td>
<td>0.03</td>
</tr>
</tbody>
</table>

In practice, cross validation [60] is a widely used and popular technique that evaluates the quality of models. In this method, instead of utilizing common approaches that split data set into training and testing data sets, it instead divides the data set
into $k$ folds, $k-1$ folds for building a model and one fold for the evaluating its quality. It also avoids over-fitting. This procedure is repeated $k$ times. The average of all correct classified instances in all runs indicate the accuracy of the model. This technique is helpful for a data set with a small number of instances. In this experiment, we set $k$ at 10. Furthermore, we use stratified cross validation to make sure that in each fold the number of instances for each class are roughly equal.

### 3.4.2 Parameter Tuning for Machine Learning Algorithms

We use the Weka toolkit (Waikato Environment for Knowledge Analysis) \[61\] for the implementations of the machine learning approaches. We use the default parameter values for the machine learning techniques in Weka. They are set as follows:

**C4.5 Decision Tree**: We use J48 which is the implementation of C4.5 Decision Tree algorithm in the Weka Toolkit. In our experiment, the $\text{confidenceFactor}$ is set at 0.25. The larger the value means more pruning and vice versa. Also, we set the number of instances per leaf at 10.

**Random Forest**: For Random Forest, we set the number of constructed trees at 100. The tree can grow, and its maximum length is not limited.

**Naive Bayes**: We set $\text{useSupervisedDiscretization}$ (that converts continuous variables into discrete or normal variables as false. For the defined features, we do not need this conversion. Furthermore, Naive Bayes does not need to tune any numeric parameters.

**Logistic Regression**: We leave the default value for the $\text{maxBoostingIterations}$ (i.e., maximum iteration for LogitBoost\(^7\)) and $\text{heuristicStop}$ (i.e., if heuristicStop > 0, LogitBoost is a boosting based classification algorithm, it starts with making weak classifiers and then makes a strong one

\(^7\)LogitBoost
the heuristic for greedy stopping while cross-validating the number of LogitBoost iterations is enabled) 500 and 50, respectively. Also, the useCrossValidation is set as true [62].

3.4.3 Evaluation Metrics

The effectiveness of the classifiers is evaluated based on four common evaluation measures: accuracy, precision, recall, and F-measure. All the mentioned metrics are based on TP (True Positive), TN (True Negative), FP (False Positive) and FN (False Negative) concepts. They are defined as follows:

- **TN**: The number of vulnerabilities classified as BV, and they are truly BV.
- **FN**: The number of vulnerabilities classified as BV, while they are truly MV.
- **FP**: The number of vulnerabilities classified as MV, while they are truly BV.
- **TP**: The number of vulnerabilities classified as MV, and they are truly MV.

The accuracy, precision, recall and F-measure for each class are calculated. The descriptions of them are as follows:

**Accuracy**: Accuracy indicates the correct classification rate. In our research, it represents the number of both BV and MV are correctly classified out all vulnerabilities.

\[
Accuracy = \frac{TP + TN}{TP + TN + FP + FN}
\] (3.3)

**Precision**: Precision is related to correctness. It can measure the efficiency of the classifier. We define it for both BV and MV as follows:
• Precision of BV: The proportion of correctly classified BV to all classified vulnerabilities as BV.

\[ P(BV) = \frac{TN}{TN + FN} \]  

(3.4)

• Precision of MV: The proportion of correctly classified MV to all classified vulnerabilities as MV.

\[ P(MV) = \frac{TP}{TP + FP} \]  

(3.5)

Recall: Recall is also known as the Probability of Detection (PD). It is the proportion of correct classification of vulnerabilities to a specific category to all actual vulnerabilities of that category.

• Recall of BV: BV recall is the total number of correctly classified BV. Equation 3.6 calculates the recall for BV.

\[ R(BV) = \frac{TN}{TN + FP} \]  

(3.6)

• Recall of MV: MV recall is the total number of correctly classified MV. It demonstrates the rate of MV detection. Equation 3.7 is used to calculate recall for MV respectively.

\[ R(MV) = \frac{TP}{FN + TP} \]  

(3.7)

The precision and recall are significant factors to measure the quality of a classification approach. A defect classification tool with low precision will not be used by developers because of high FP (a large number of actual MVs are classified as
3.4. EXPERIMENTAL EVALUATION AND RESULTS

BVs). Also, a tool with low recall means MV/BV has not been predicted accurately. Achieving high recall and precision at the same time is not practical. There is a trade-off between them. The recall should be sacrificed to get high precision and vice versa [63]. We use F-measure instead of precision and recall.

**F-measure**: F-measure indicates the harmonic mean of precision and recall. It shows how an increase in precision leads to a decrease in recall. In this work, a classifier with the highest MV F-measure on the data set is selected. In a large number of published papers in software engineering, the F-measure is considered for final judgement [35,60]. Then it is used to predict the category of unseen vulnerabilities. F-measure formula for BV and MV are given in Equation 3.8 and 3.9.

\[
F(BV) = \frac{2 \times P(BV) \times R(BV)}{P(BV) + R(BV)} \quad (3.8)
\]

\[
F(MV) = \frac{2 \times P(MV) \times R(MV)}{P(MV) + R(MV)} \quad (3.9)
\]

3.4.4 Results and Discussions

In this section, the discussed evaluation metrics including Accuracy, R(BV), R(MV), P(BV), P(MV), F(BV), and F(MV) are calculated for the machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) as shown in Table 3.6.

The columns represent the mentioned evaluation metrics and the rows indicate the value of metrics for the utilized machine learning algorithms. Since there is no related work on vulnerability classification based on security failure reproducibility, we compare the results with research on defect classification into the BB and MB.
Table 3.6: Comparison between different classifiers

<table>
<thead>
<tr>
<th>Alg</th>
<th>Acc</th>
<th>R(BV)</th>
<th>R(MV)</th>
<th>P(BV)</th>
<th>P(MV)</th>
<th>F(BV)</th>
<th>F(MV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>67</td>
<td>60.5</td>
<td>73.2</td>
<td>69.3</td>
<td>64.9</td>
<td>64.5</td>
<td>69</td>
</tr>
<tr>
<td>RF</td>
<td>65</td>
<td>64.7</td>
<td>65</td>
<td>64.9</td>
<td>64.8</td>
<td>64.8</td>
<td>65</td>
</tr>
<tr>
<td>LR</td>
<td>60.5</td>
<td>61.8</td>
<td>59.2</td>
<td>60.6</td>
<td>60.1</td>
<td>60.1</td>
<td>55.9</td>
</tr>
<tr>
<td>NB</td>
<td>56.8</td>
<td>80.1</td>
<td>36.6</td>
<td>55.8</td>
<td>64.7</td>
<td>65.8</td>
<td>46.8</td>
</tr>
</tbody>
</table>

The USES technique (as mentioned in Section 2.4) analyses defects based on natural language description [35]. The calculated F-measure scores of the paper [35] are between 29.8% and 61.5% based on textual bug reports, while our approach gains MV F-measure score in the range of 46.8% and 69% regardless of the type of used machine learning approaches.

The main reason for selecting different machine learning techniques is to show the effectiveness of the metrics selected for this research. We do not intend to prove which algorithm is the best. How accurate a classifier can distinguish BVs from MVs is an important factor. The accuracy of all models is more than 57%. It indicates the efficiency of defining metrics based on both textual reports and code fixes to classify BVs and MVs.

The framework can classify security vulnerabilities into MVs in most cases even if some BVs are classified as MVs incorrectly. We believe that in vulnerability classification, considering BV as MV mistakenly is better than not detecting MVs, while they can remain in the system and cause serious security failures. From this result, we can conclude that the Decision Tree approach obtains the better results than other approaches in terms of accuracy and MV F-measure scores. The main reason is related to its simplicity and robustness against noise in the data set. Overall, the classification framework is able to classify 65% of MVs with an accuracy of 67%.
3.5 Summary

In this chapter, the vulnerability classification framework that can classify vulnerabilities into the BV and MV categories is developed. The framework uses the textual reports, source code, and history of code fixes to define features. Then, the most discriminative features are identified based on the Fisher scores. Mozilla Firefox has been described as a case study. As features are defined based on code fixed and textual reports, the resources are integrated. Overall, 580 post-release vulnerabilities from thirty-nine releases from January 21, 2005 to January 13, 2015 are collected. After analysing the data set, we make it balanced to avoid over-fitting.

Then, we apply different machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) on the data set to build classifiers. We use the machine learning implementation with default parameters in Weka Toolkit. The results indicate that all the used machine learning techniques provide high accuracy and F-measure. We can conclude that the metrics are effective for vulnerability type identification irrespective of the type of the utilized machine learning technique.

Another interesting result is that even if the C4.5 Decision Tree is simpler than Random Forest, it can perform as effective as Random Forest in classifying BVs and MVs. As the results show, the algorithms with the default parameters generate accurate classifiers. However, the accuracy of the models can be increased by parameter optimization.
Chapter 4

A Vulnerability Prediction Framework

This chapter starts with three hypotheses to find relationships between the difficulty of the security failure reproducibility and source code. It reviews some classical software metrics (McCabe’s, Halstead’s, program size, and object-oriented) and some factors related to the environment of the system (concurrency, interaction with states of a system, runtime error, and memory management) to test the hypotheses. The process of the metric extractions for the risky files in the data set is described. The metric values are used to build multiple linear regression models. The models can investigate the potential correlations between the metrics and the target categories. An MV-prone prediction framework is developed based on the metrics with strong correlations and well-known machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes). The prediction framework is a great help for security experts to focus their effort on the top-ranked vulnerability-prone files.

4.1 Prediction Hypotheses

Making an objective measurement or generalization over vulnerability features is difficult. Several researchers have analysed computer vulnerabilities to collect them for
the software development process in the hope that they do not appear again. However, they continue to occur. Therefore, it can be inferred that a list of raw data of exploited vulnerabilities cannot cause developers to avoid them. Software developers need to have a deep understanding of vulnerability characteristics and their occurrence in order to avoid or remove them from the software development process. Specialized knowledge about the characteristics of different vulnerability types gives security engineers more insight about attacks. When software engineers have knowledge about software security, they can perform better in detecting and eliminating vulnerabilities.

To achieve information about the nature of BVs and MVs at the source code level, we postulate three hypotheses as follows:

**H1. Vulnerabilities can be classified into BVs and MVs based on the software metrics.**
In this case, MV files are files for which at least one security failure due to MV is reported by testers, end users, or third party researchers. However, files with BVs are free from any MV.

**H2. BVs have significant correlations with the software metrics.** This hypothesis focuses on files with BVs.

**H3. MVs have significant correlation with the software metrics.** This hypothesis focuses on files with MVs.

In the following sections, the hypotheses are tested.

### 4.2 Software Metrics

A general technique for vulnerability analysis is to use software metrics for building statistical models. The classical software metrics are also utilized to determine the
quality of software systems. They measure attributes of software products and development processes. Previous literature shows that the performance of vulnerability prediction models changes according to utilized metrics.

Usually, classical software metrics which are commonly used for defect prediction (such as code metrics [64], inter-modules [8], inter-developer relationship [10]) are utilized for vulnerability prediction as well. Furthermore, text mining [11, 65] and organizational structure [9] techniques are studied for the vulnerability prediction. To choose vulnerability prediction metrics, there are two solutions: the classical metrics (that perform very well for the defect prediction) and new metrics (that can have potential relationships with vulnerability reproducibility). We use both of them.

4.2.1 Classical Software Metrics

Some studies adopt the classical software metrics for classification and prediction. They show that the metrics are effective to identify post-release defects rather than the pre-release (vulnerabilities occurred in the testing phase) ones.

Security experts believe that complexity can decrease the level of the system security. The complicated code causes vulnerabilities whose diagnosis, testing, and maintenance are really hard [66]. This condition can increase the likelihood of attacks since attackers exploit the vulnerabilities in the source code.

The complexity of software can be related to the complexity of four factors: problem, algorithm, structure, and cognition [67]. This research focuses on structural complexity, though all factors can be related. We hypothesize that classical software metrics (McCabe's cyclomatic complexity [68], Halstead's [69], program size, and object-oriented metrics) might be related to the presence of BVs and MVs because
### 4.2. SOFTWARE METRICS

Table 4.1: Software complexity metrics

<table>
<thead>
<tr>
<th>Type</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe's cyclomatic complexity</td>
<td>AvgCyclomatic, CyclomaticModified, CyclomaticStrict, Essential, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential, MaxNesting</td>
</tr>
<tr>
<td>Halstead's metrics</td>
<td>n1, n2, N1, N2, Vol, Len, Voc, Dif, Eff</td>
</tr>
<tr>
<td>Object-oriented</td>
<td>CountClassBase, CountClassCoupled, CountDeclClass, CountDeclMethodDefault, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethodProtected, CountClassDerived, CountDeclMethodPublic, CountDeclClassVariable, MaxInheritanceTree, CountDeclMethod, CountDeclMethodAll, CountDeclClassMethod, CountDeclMethodPrivate, PercentLackOfCohesion</td>
</tr>
</tbody>
</table>

These metrics yield an important representation of complexity. The metrics can be calculated during various phases of software development such as coding and design. Furthermore, they are very helpful for quality evaluation [70]. Table 4.1 summarizes the metrics that belong to each category. The descriptions of the metrics can be found in [70].

**McCabe's cyclomatic complexity**: McCabe's cyclomatic complexity measures the complexity of a program based on the number of non-dependent linear paths [68]. We collect different types of McCabe's cyclomatic complexity including modified and strict cyclomatic complexity. *MaxNesting* calculates the maximum number of control
4.2. SOFTWARE METRICS

statements (e.g., if and loop) which are nested inside each other. Control structures
which are highly nested can increase the complexity of program units and decrease
the understandability of the program.

**Halstead's software science:** Maurice Halstead proposed the Halstead's metrics
in 1970s. Halstead is adopted in industry and universities. It calculates the number
of operators and operands [69].

**Program size:** The program size metrics depict the lines of code (LOC), declara-
tions, statements, and files.

**Object-oriented (OO):** The quality of software systems can be measured by the
object-oriented metrics. The traditional software metrics are not developed for object-
oriented programs (OOP). Moreover, the OO metrics are not thoroughly studied
in order to understand if they can measure the complexity of this program type.
In this research, we collect a set of metrics related to the coupling, encapsulation,
polymorphism, and inheritance [71] for our object-oriented based case study. Some
of the important object-oriented metrics are described as follows:

1. Coupling: Stevens et al. propose a definition for coupling as "the measure of the
strength of association established by a connection from one module to another" [72].
For instance, there are two objects, *obj1* and *obj2* from different classes. If *obj1* calls
a function or uses a variable from *obj2*, two classes are coupled. Also, two entities
can be coupled. The coupling can estimate how complicated an information flow is.

2. Lack of Cohesion in Methods (LCOM): Cohesion in a class indicates how closely
methods are related to local variables. A program with low cohesion can make a
program complicated.

3. Weighted Methods per Class (WMC): WMC enumerates the number of methods
in each single class. A class with a large number of functions is complicated. It increases the probability of error presence. It can influence child nodes because they can inherit functions from their parents.

4. Inheritance: Inheritance increases the level of abstraction. As a result, it decreases the number of operands and operations. It helps have a program with low complexity. The depth and breadth of an inheritance hierarchy can be calculated based on two metrics:

- Depth of Inheritance Tree (DIT): DIT is defined as the maximum number of nodes between the class and the root node of the tree. A deep class increases the complexity and fault-proneness of the program because it can inherit more functions.

- Number of Children (NOC): The NOC metric measures subclasses. When a considerable number of nodes have the same parent, changing the parent class is difficult due to its impact on the children. To service child nodes, it should be more flexible. This condition increases the complexity of the program.

4.2.2 Metrics for Vulnerability Reproducibility

We assume that the reproducibility of some security failures can be influenced by runtime environmental factors which include operating systems and other applications. To introduce a set of metrics which can have direct or indirect relationships with vulnerability reproducibility, we review some defect analysis techniques even though they do not accept the BB and MB terms [23, 34, 73, 74].

We consider four main factors: concurrency, interaction with the system environment (i.e., anything that is not inside of the program), runtime error, and memory
management. There might be more metrics that are relevant to the reproducibility, but we focus on these categories in this research. A summary of them can be found in Table 4.2.

Table 4.2: Factors based on the internal environment of the system

<table>
<thead>
<tr>
<th>Type</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>The number of threads and the number of synchronized method</td>
</tr>
<tr>
<td>Program interaction</td>
<td>Input/output libraries</td>
</tr>
<tr>
<td>Runtime error</td>
<td>The number of try-catch statements</td>
</tr>
<tr>
<td>Memory management</td>
<td>The number of alloc, dealloc, malloc, free, and delete</td>
</tr>
</tbody>
</table>

*Concurrency:* In multi-threaded programs, a specific scheduling of a thread may cause a transient security failure. We count the number of threads and synchronized methods in the files. Although it is possible to use more metrics (e.g., the amount of concurrent execution), their measurements are hard [75]. We do not consider them in order to keep the approach simple.

*Interaction of the program with the environment of the system:* The environment of the system running the program (that includes hardware and applications executing on the system) can influence the MV activation. When interaction is not performed correctly, it increases the possibility of MVs in the system. In programs, most of the interactions are done via I/O interfaces. We check if I/O library (e.g., `<iostream>`) is called in a file. The value of this metric is 0, 1. If an I/O library is used in a file, the metric is set at 1, otherwise, it is set at 0.

*Runtime error:* Complicated programs have more potential to errors. Some errors happen at runtime because they are triggered by a specific event or internal state of the system (which is similar to the MV activation). They are captured by the
exception handling. In the try-catch, try checks the block for errors and catch handles it. We count the number of times the try-catch are used in the files.

Memory management: The experiment (Section 3.3) shows that the majority of the MVs are accumulated in the memory. Memory related issues can cause memory leaks or memory corruption. They can increase failure rate and decrease the performance of the system. Different memory-related factors may impact the reproducibility of security failures. However, dynamic memory allocation and deallocation are important factors from our perspective. The main reason of memory faults is a direct access to the memory for allocation and deallocation done by C and C++. For each file, the number of dynamic memory allocation (e.g., alloc and malloc) and deallocation (e.g., free and dealloc) are counted.

4.3 Metric Extraction for Code Analysis

In this research, we analyse security issues at the file level instead of component or version levels because there exist a large number of files in Mozilla Firefox, and they are too large for static analyses. An overview of the proposed approach for correlation investigation between the metrics and the target categories is shown in Figure 4.1. The risky files extracted in Section 3.2.3 are studied. They are passed to the Metric Extractor component shown in Figure 4.2. It uses Understand C++¹, a third-party commercial tool to extract software metrics. This tool is selected because it is user-friendly, and it is able to automatically extract software metrics. More importantly, it enables users to run Perl and Python APIs for additional metrics.

Some metrics are computed at the class or function level instead of the file level

¹http://www.scitools.com
4.3. METRIC EXTRACTION FOR CODE ANALYSIS

Figure 4.1: An overview of the correlated metrics investigation for BVs and MVs

Figure 4.2: Metric Extractor component

(e.g., CountClassCoupled, PercentLackOfCohesion, MaxInheritanceTree, and CountClassDerived), and the files may contain more than one class. In this case, the metrics are computed for each class or functions in a file, and they are averaged to get a number representing the whole file. To extract more metrics, we use Perl APIs to query the database for the metrics. For each category, the APIs mine the source code and look for keywords that correspond to the metrics. For example, for dynamic memory allocation, we search for keywords 'alloc' and 'malloc', and count the number of times they appear. We focus on the source files (i.e., C, C++, and header files), and exclude other files (e.g., scripts, configuration, and make) from our analysis.

Then, the #BV & #MV per File Identifier component counts how many times each file have been changed to address security failures related to the BV and MV categories. Finally, the files that are represented by feature vectors are ready for the
analysis. The Metric Correlation Investigator component is presented in the following section to identify the correlation between the metrics and BV and MV categories.

4.4 Metric Correlation Investigation

To make the descriptions of the categories simple, a multiple linear regression technique is adopted. A multiple linear regression technique builds a model with more than one independent variables (is called explanatory or predictor) and the dependent variable (explains the class).

Before building a multiple regression model, we need to make sure that each independent variable has a significant correlation with the dependent variable. Hence, we first investigate the simple linear correlations between each independent and dependent variables. The procedure is described in the following subsection.

In this chapter, statistical analysis is done with the help of IBM SPSS software\(^2\).

**Linear Correlation Coefficient**

An obvious approach to find the quantitative relationships between the metrics and the target categories is to build a regression model. The linear regression technique builds a model with one independent variable (the metrics) and the dependent variable (the BV and MV categories). This technique selects each independent variable, individually. Then, to measure the significance of correlations between the metrics and the categories, Pearson correlation coefficient \((r)\) and \(p\)-value (i.e., the possibility of test-statistic) are computed. Pearson correlation coefficient \((r)\) measures the strength of a correlation. It is \(-1 \leq r \leq 1\). A positive and negative \(r\) denote positive

and negative linear correlations, respectively. A correlation can be weak \((0 \leq r < 0.3)\), moderate \((0.3 \leq r < 0.5)\), or strong \((r \geq 0.5)\) [8].

To determine if a correlation is not accidentally observed, \textit{p-value} (i.e., the possibility of test-statistic) is used. Traditionally, \textit{p-value} \(\leq 0.05\) is significant. It means that we are 95\% confident that the correlation is not accidentally observed. The higher the correlation, the better the predictor. The correlation between the metrics (see Section 4.2) and the hypotheses \(H1\), \(H2\), and \(H3\) are investigated. The results of the experiments are shown in Table 4.7

\(H1\): As the results indicate in the second column of the table, all the metrics show low \(r\) \((r < 0.3)\) with high \textit{p-value} \((p-value > 0.05)\). Any significant linear correlation between the software metrics and the categories is not observed. It rejects hypothesis \(H1\) that the reproducibility of security failures can be recognized at the code level. Hence, we can conclude that the code archives and textual vulnerability reports are resources for the vulnerability classification. To get insight into the natures of BVs and MVs from the source code perspective, we analyse the BV and MV categories separately.

\(H2\): On average, the \(r\) values for the metrics with significant correlations are between 0.041 and 0.436. According to the \(r\) values, some correlations are weak, while those of others are moderate. The results confirm our hypothesis that there is a linear correlation between the software metrics and presence of BVs. The moderate correlations are related to \textit{CountLineCode}, \textit{CountLineComment}, \textit{CountLineBlank}, and \textit{CountLineInactive} (from the preprocessor perspective, the lines that are not on the True parts of a \#if or \#ifdef) [70]. BVs have a weak relationship with the complexity of the program structure (e.g., McCabe’s and Halstead’s), and BV activation is not
influenced by the environmental conditions. Consequently, the reproduction process of security failures due to BV is simple. In this class, among the metrics with significant correlation, \textit{CountLineComment} (0.436) and \textit{MaxEssential} (0.006) are the best and the worst indicators for BVs based on the \( r \) values, respectively. It indicates that simple but large programs are more vulnerable to BVs.

\textit{H3:} The \( r \) values of the metrics with significant correlations (\textit{p-value} < 0.05) are in the range of 0.25 and 0.53. The Halstead's metrics, the number of lines of code, blank, and comment, McCabe's cyclomatic complexity, and dynamic memory allocation indicate moderate and significant correlations with the number of MVs. Based on the results, complicated large code has potential to security failures with a hard reproduction process. Among all the metrics with significant correlations, \textit{CountLinePreprocessor} gets the highest \( r \) value, while \textit{CountStmtEmpty} gains the lowest \( r \) value in this category.

The program size metrics are considered as effective defect predictors. In our research, they show significant relationships with the BV and MV categories as well. On average, the \( r \) values of the metrics for the MV category are higher than for the RV category. In addition, we conclude that the object-oriented metrics are less likely related to the vulnerability presence.

The correlations might vary according to the project under analysis. For instance, the complexity of PostgresSQL (i.e., a relational database management system (DBMS)) is more correlated to the comments per line of code [28]. In contrast, this metric shows poor correlations in our case study. In the following subsection, the multiple linear regression models are constructed based on the identified metrics with strong correlations.
4.4.4 Metric Correlation Investigation

4.4.1 Multiple Linear Regression Model

The multiple linear regression model formula is as follows: \( \hat{Y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + ... + \beta_n x_n + \epsilon \), in which \( x_i \) are independent variables, \( \beta_i \) are linear parameters, \( n \) is the number of observation, \( \epsilon \) is a random error, and the \( \hat{Y} \) is the expected dependent variable.

There are correlations between some software metrics (e.g., a large number of lines of code can lead to a large number of functions) [76]. Metrics with inner correlations should not be used as a model at the same time because any variation in the data set might cause large differences in the model [77].

To develop a stable model, a stepwise regression method [76] is used to select the most appropriate independent variables. In this method, different features are added and deleted from a model until no further improvement can be achieved. It chooses independent variables with high \( r \) values with the dependent variable. The qualities of models are assessed based on two criteria: significance of coefficient (\( p\)-value) and goodness of fit (\( R^2 \)). \( R^2 \) can measure the variation in the data set. It demonstrates how strong a predictive model is. The \( R^2 \) formula is as follows:

\[
R^2 = \frac{\sum(Y_i - \bar{Y})^2}{\sum(Y_i - \bar{Y})^2}
\]

where \( \bar{Y} \) is a mean value, \( Y_i \) is a sample value, and \( \bar{Y} \) is a model value. The results of the multiple linear regression models are indicated in Table 4.5.

\( H2 \): A stepwise regression model identifies three variables. The model achieves \( R^2 \) of 0.319. It demonstrates that the number of BVs is more correlated with the comment lines, inactive line, and empty statements.

\( H3 \): In the MV case, the stepwise regression method builds a model according to the four variables. The stepwise model demonstrates that on average, security issues in a program with a large number of functions, preprocessors, empty statements, and
dynamic memory allocations have a difficult reproduction process. The model gets $R^2$ of 0.47. We then employ principal component analysis approach that is described in the following paragraph. We hope that it can improve the accuracy of the model.

**Principal Component Analysis (PCA)**

A large number of metrics can increase the probability of multicollinearity [76]. There are different techniques in the factor analysis field. To address the multicollinearity issue [76,78], some studies adopt the Principal Component Analysis (PCA) approach [76] (i.e., a well-known dimensionality reduction technique).

The PCA takes the data set with many dimensions’ space, then, flattening the data sets into a smaller number of uncorrelated variables named principal components. It attempts to maximize the variance in the data. The PCA algorithm identifies components ($Z = [Z_1, \cdots, Z_m]$), that are the linear combination ($[u_1, \cdots, u_m]'$) of the variables ($X = [X_1, \cdots, X_m]$) $Z = UX$. It uses the data in a new coordinate system that enlarges variations of dimensions ($u'u = 1$), while keeping them uncorrelated. The first component ($Z_1$) indicates the maximum variance. The information which is not taken by the first component and is uncorrelated to it is identified by the second component ($Z_2$) and so on.

To find components, the eigenvalue (i.e., eigenvalue $\lambda$ indicates that if there is a specific matrix $z$ for the matrix $A$ providing that $Az = \lambda z$ is true) decomposition of the correlation matrix of variables is used. Principle components for the BV and MV categories are extracted, the results are shown in Table 4.3 and Table 4.4, respectively. Components with Eigenvalue less than 1 have low variance, and they are not kept for

---

3These components are different from the components of the classification and prediction frameworks.
further analyses. The PCA technique reduces the number of dependent variables to
the eight uncorrelated variables that are responsible for 84.88% of total variance in
the files with BVs. Also, for the MV case, 85.98% of total variance is related to the
eight components. We build multiple regression models for each category according
to the eight components. The models get $R^2$ of 0.27 and 0.3 for the BVs and MVs,
respectively.

Table 4.3: Dimensionality reduction by PCA for BV

<table>
<thead>
<tr>
<th># of component</th>
<th>Proportion of explained variation</th>
<th>Total</th>
<th>% of Variance</th>
<th>Cumulative%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.9</td>
<td>45.4</td>
<td>45.4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11.36</td>
<td>19.9</td>
<td>65.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.4</td>
<td>5.9</td>
<td>71.2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.8</td>
<td>74.9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.8</td>
<td>3.1</td>
<td>78.0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.5</td>
<td>2.6</td>
<td>80.6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.3</td>
<td>2.3</td>
<td>82.9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.1</td>
<td>1.9</td>
<td>84.9</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Dimensionality reduction by PCA for MV

<table>
<thead>
<tr>
<th># of component</th>
<th>Proportion of explained variation</th>
<th>Total</th>
<th>% of Variance</th>
<th>Cumulative%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.5</td>
<td>51.7</td>
<td>51.7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9.8</td>
<td>17.3</td>
<td>68.9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.7</td>
<td>4.8</td>
<td>73.8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.6</td>
<td>3.6</td>
<td>77.4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
<td>2.7</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.2</td>
<td>2.2</td>
<td>82.2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.1</td>
<td>1.9</td>
<td>84.2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1.8</td>
<td>85.9</td>
<td></td>
</tr>
</tbody>
</table>

To identify uncorrelated metrics with high variance, the scatter plot of data is
really helpful. For the BV and MV categories, Figure 4.3(a) and Figure 4.3(b) indi-
cate that before the elbow (circle) of the figures there are two components that are
4.4. METRIC CORRELATION INVESTIGATION

Figure 4.3: Screen plot of PCA.

The metrics selected by the stepwise methods get higher $R^2$ meaning that they are suitable indicators for predicting BVs and MVs. Nonetheless, the models may not be the best fit for our data set. They provide information about the nature of BVs and
MVs. These achievements encourage security experts to use the metrics with high correlations at the earlier phases to cope with security failures with a complicated reproduction process. As a large number of residual vulnerabilities in the system belong to the MV (Section 3.3), we propose an MV-prone prediction framework in the next section to classify vulnerable files as MV prone or non MV-prone.

4.5 MV-prone Prediction Framework

Research in the software vulnerability prediction is classified into two main categories:

1. Count-based technique: It anticipates how many vulnerabilities reside in an entity (e.g., function, class, file, and component).

2. Classification-based technique: It predicts entities that are prone to vulnerabilities in a software application. Although the main goal of vulnerability prediction is to avoid the vulnerability presence in the next versions, they are used for reactive activities.

In this research, files with MV-proneness are predicted rather than anticipating the number of MVs in the files. Vulnerable prone identification guides software engineers to allocate resources by considering vulnerable-prone entities. The main goal of the model is to decrease the MV presence in the next releases of the software. The model can provide security engineers with additional information. It assists security experts to focus and prioritize their effort according to the MV-proneness of files.

An overview of the proposed MV-prone prediction framework is shown in Figure 4.4. The Metric Extractor component extracts the value of the four identified metrics for the risky files with MVs. Then, the MV-prone Model Builder employs the machine learning techniques to construct predictors.
4.5. MV-PRONE PREDICTION FRAMEWORK

The output of the classification framework (Chapter 3) is analysed. If the category of an unseen vulnerability is identified as MV, the Metric Extractor component extracts the value of the four metrics for the related risky files. Then the MV-prone Predictor component gets the predictor and the values of the metrics to identify if the files are MV-prone or not. The prediction framework components are described in the following subsections in more details.

4.5.1 Metric Extractor

Selecting appropriate metrics is one of the most important steps before adopting any machine learning technique. Metrics with significant correlations are effective for building high performance predictive models. Section 4.4 indicates that the
4.5. MV-PRONE PREDICTION FRAMEWORK

Four metrics (\textit{CountDeclFunction}, \textit{alloc}, \textit{CountStmtEmpty}, and \textit{CountLinePreprocessor}) are the most significant correlated metrics with the number of MVs. The higher the correlation, the better the predictor. This component computes the values of the four metrics for each file in the MV category.

4.5.2 MV-prone Model Builder

We propose five steps to identify MV-prone and non MV-prone files:

\textit{Step 1. Order the files with respect to the number of MVs.} It is believed that a large proportion of vulnerabilities in the next versions are related to the vulnerable files in the previous versions.

\textit{Step 2. Select metrics with strong correlations with the number of MVs.} Selecting appropriate features is one of the most important steps before adopting any machine learning technique. The correlation investigation between the metrics and MVs are presented in Section 4.4.

\textit{Step 3. Label the top 20\% of the files as MV-prone, and the rest of the files as non MV-prone based on the '80-20' rule of thumb.} The experiment indicates that a small number of the vulnerable files are responsible for a large number of security failures in this category [42]. In our case study, 25\% of the MVs belong to 7\% of the vulnerable files. Furthermore, 45\% of the security failures in the MV category involve 20\% of the files with security vulnerabilities. This result is consistent with the results of the most studies in fault-prone prediction models.

\textit{Step 4. Make the data set balanced to eliminate over-fitting} [2]. We make it balanced according to the approach presented in Section 3.4.1 to avoid over-fitting.

\textit{Step 5. Employ different machine learning techniques to find the best predictor.} In
most of the previous work, Logistic Regression [29] has been used for the vulnerability prediction as it performs very well for the binary classification. In addition to Logistic Regression, we employ four statistical techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes). The default parameter values (mentioned in Section 3.4.2) are used for the implementation of the machine learning techniques in Weka. The results are discussed in Section 4.5.4.

4.5.3 MV-prone Predictor

The MV-prone Predictor takes two inputs: the prediction model and the output of the vulnerability classification framework (proposed in Chapter 3). When the output of the classification is MV, the relevant risky files are passed to the Metric Extractor component to calculate the four significant correlated metrics. The MV-prone Predictor predicts if the relevant files are MV-prone or not.

4.5.4 Evaluation and Results

To assess the success of the vulnerability prediction models, we use well-known evaluation metrics: accuracy, precision, recall, and F-measure. They are defined for the classification in Section 3.4.3. In this chapter, we redefine them for the prediction. Also, the FP rate is an important performance indicator [75] for the vulnerability prediction.

Accuracy: Accuracy is the proportion of correct prediction of both MV-prone and non MV-prone files to all files. Achieving relatively high accuracy is important for a
4.5. MV-PRONE PREDICTION FRAMEWORK

model, but not sufficient necessarily.

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (4.1)
\]

**Precision**: Precision is the proportion of the correct prediction of vulnerable files as MV-proneness to all predicted files as this category. It is an important performance indicator to measure the effectiveness of the predictors.

\[
\text{Precision} = \frac{TP}{TP + FP} \quad (4.2)
\]

**Recall**: Recall is the proportion of correct prediction of MV prone files to all actual MV-prone files. A model with high PD indicates its ability to discover more vulnerabilities. In security research, PD is considered as the most important performance indicator [10, 37].

\[
\text{Recall} = \frac{TP}{FN + TP} \quad (4.3)
\]

**F-measure**: If precision and recall of a vulnerability prediction tool are low, security experts are reluctant to use it due to its inability to predict the category of vulnerabilities accurately.

\[
\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4.4)
\]

**FP rate**: FP rate is the proportion of incorrect prediction of a non MV-prone file to all actual files in this category. When the probability of the FP is high, developers inspect and test the file excessively- however, this level of efforts is not required [75].
4.5. MV-PRONE PREDICTION FRAMEWORK

The FP rate has an inverse relationship with precision.

\[ FPrate = \frac{FP}{TN + FP} \] (4.5)

Evaluating the models based on all the mentioned evaluation metrics may increase redundancy since some of them are related. In this research, we compare the predictors based on the accuracy, recall, F-measure, and FP rate. These metrics are selected because they are used in the previous paper [57]. The prediction results are shown in Table 4.6. We extract some interesting results as follows:

Table 4.6: Comparison of different machine learning techniques for the MV-prone prediction framework

<table>
<thead>
<tr>
<th>Alg</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
<th>FP rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>75</td>
<td>83.5</td>
<td>84.2</td>
<td>83.3</td>
<td>8.4</td>
</tr>
<tr>
<td>LR</td>
<td>64</td>
<td>81.4</td>
<td>82.5</td>
<td>79.4</td>
<td>2.8</td>
</tr>
<tr>
<td>DT</td>
<td>60</td>
<td>73</td>
<td>74</td>
<td>68</td>
<td>14.7</td>
</tr>
<tr>
<td>NB</td>
<td>59</td>
<td>74</td>
<td>77</td>
<td>71.1</td>
<td>9.1</td>
</tr>
</tbody>
</table>

1. The accuracy of the models is higher than 59%. This level of accuracy indicates that the metrics developed for non-security purposes can be effective for predicting MV-prone files.

2. Logistic Regression is one of the widely used algorithms for the vulnerability prediction. However, our results imply that Random Forest can predict MV-prone files with higher accuracy, recall, and a low number of false positive than Logistic Regression. Random Forest can provide an accurate predictor as it removes noise from the data set.

3. Predicting MV-prone files is really important for a prediction model even if some non MV-prone files are mistakenly predicted as MV-prone (False Positive). In fact,
4. **SUMMARY**

A model can be suitable even if it does not have small number of false positives.

4. As far as there is no previous work related to the MV-prone prediction, we compare our results with vulnerability prediction approaches. Shin et al. [64] propose a vulnerability prediction model based on the Logistic Regression algorithm for Mozilla Firefox. Although the model in this paper is able to predict vulnerabilities with a very small false positive (0.74%), its low recall (13.2%) is not acceptable. We anticipate that biasing toward the class with a major number of data samples can be a possible reason. It can impact the effectiveness of the model. We first make the data set balanced then we train the predictors. In our case, Logistic Regression can anticipate MV-prone files with 71% recall and 22% false positives. Although the results are desirable, Random Forest and Decision Tree are able to generate predictors with higher performance.

5. The prediction framework can predict 83% of MV-prone files with 75% accuracy.

### 4.6 Summary

Knowing more about the nature of BVs and MVs from the source code perspective can guide security experts to address security issues in the earlier stages of the software development. In this chapter, three main hypotheses based on the security failure reproducibility and the source code are made as follows: 

**H1.** Vulnerabilities can be classified into BVs and MVs based on the metrics.

**H2.** BVs have linear correlations with the software metrics.

**H3.** MVs have linear correlations with the software metrics.

In this research, we use the classical software metrics and a set of runtime metrics (that are defined based on the definition of MVs). Then, we use Understand C++ to automatically extract the metric values for the relevant vulnerable files of Mozilla.
4.6. SUMMARY

Firefox. This information is used to test the hypotheses. For each hypothesis, we build multiple linear regression models with the help of IBM SPSS. The results are summarized as follows:

- **H1**, all the considered complexity metrics show poor linear correlations with the target categories in the binary classification (BVs and MVs). Hence, the software metrics cannot distinguish BVs from MVs. As shown in Chapter 3, other resources (such as textual reports and code fixes) can be more effective for the binary classification.

- **H2**, the number of inactive lines and lines having comments and empty statements is significantly related to the number of BVs.

- **H3**, the number of MVs have significant correlations with the number of functions, preprocessor, dynamic memory allocation, and the number of empty statements.

We notice that the object-oriented metrics are less likely related to vulnerability presence for both categories (the hypotheses **H2** and **H3** \( p\text{-value} < 0 \)). After identifying the strongly correlated metrics with MVs, an MV-prone prediction framework is developed. Similar to the training phase of the classification framework, the four well-known machine learning techniques (C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes) are employed to find the best predictor. We use Weka Toolkit for machine learning implementations with default values for the parameters.

The accuracy of the models is higher than 59% that demonstrates the effectiveness of the selected metrics for the MV-prone prediction regardless of the type of used
4.6. SUMMARY

Table 4.7: Pearson correlation for the hypotheses $H1$, $H2$, and $H3$

<table>
<thead>
<tr>
<th>Metrics</th>
<th>$H1$</th>
<th>$H2$</th>
<th>$H3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r$ (p-value)</td>
<td>$r$ (p-value)</td>
<td>$r$ (p-value)</td>
</tr>
<tr>
<td>n1</td>
<td>-0.008 (p &lt; 0)</td>
<td>0.248 (p &lt; 0.001)</td>
<td>0.513 (p &lt; 0.001)</td>
</tr>
<tr>
<td>n2</td>
<td>0.02 (p &lt; 0)</td>
<td>0.347 (p &lt; 0.001)</td>
<td>0.498 (p &lt; 0.001)</td>
</tr>
<tr>
<td>N1</td>
<td>0.027 (p &lt; 0)</td>
<td>0.360 (p &lt; 0.001)</td>
<td>0.449 (p &lt; 0.001)</td>
</tr>
<tr>
<td>N2</td>
<td>0.017 (p &lt; 0)</td>
<td>0.349 (p &lt; 0.001)</td>
<td>0.457 (p &lt; 0.001)</td>
</tr>
<tr>
<td>Len</td>
<td>0.023 (p &lt; 0)</td>
<td>0.356 (p &lt; 0.001)</td>
<td>0.453 (p &lt; 0.001)</td>
</tr>
<tr>
<td>Voc</td>
<td>0.008 (p &lt; 0)</td>
<td>0.327 (p &lt; 0.001)</td>
<td>0.507 (p &lt; 0.001)</td>
</tr>
<tr>
<td>Vol</td>
<td>0.033 (p &lt; 0)</td>
<td>0.371 (p &lt; 0.001)</td>
<td>0.435 (p &lt; 0.001)</td>
</tr>
<tr>
<td>Dif</td>
<td>-0.009 (p &lt; 0)</td>
<td>0.370 (p &lt; 0.001)</td>
<td>0.475 (p &lt; 0.001)</td>
</tr>
<tr>
<td>Eff</td>
<td>0.021 (p &lt; 0)</td>
<td>0.349 (p &lt; 0.001)</td>
<td>0.342 (p &lt; 0.001)</td>
</tr>
<tr>
<td>MI</td>
<td>0.067 (p &lt; 0)</td>
<td>-0.022 (p &lt; 0)</td>
<td>-0.098 (p &lt; 0)</td>
</tr>
<tr>
<td>AltAvgLineBlank</td>
<td>-0.029 (p &lt; 0)</td>
<td>0.102 (p &lt; 0.1)</td>
<td>0.102 (p &lt; 0.1)</td>
</tr>
<tr>
<td>AltAvgLineCode</td>
<td>-0.031 (p &lt; 0)</td>
<td>0.041 (p &lt; 0.1)</td>
<td>0.041 (p &lt; 0.1)</td>
</tr>
<tr>
<td>AltAvgLineComment</td>
<td>-0.052 (p &lt; 0)</td>
<td>0.053 (p &lt; 0.1)</td>
<td>0.053 (p &lt; 0.1)</td>
</tr>
<tr>
<td>AltCountLineBlank</td>
<td>0.048 (p &lt; 0)</td>
<td>0.424 (p &lt; 0.001)</td>
<td>0.424 (p &lt; 0.001)</td>
</tr>
<tr>
<td>AltCountLineCode</td>
<td>0.022 (p &lt; 0)</td>
<td>0.43 (p &lt; 0.001)</td>
<td>0.43 (p &lt; 0.001)</td>
</tr>
<tr>
<td>AltCountLineComment</td>
<td>0.037 (p &lt; 0)</td>
<td>0.369 (p &lt; 0.001)</td>
<td>0.369 (p &lt; 0.001)</td>
</tr>
<tr>
<td>AvgCyclomatic</td>
<td>0.014 (p &lt; 0)</td>
<td>0.064 (p &lt; 0)</td>
<td>0.021 (p &lt; 0)</td>
</tr>
<tr>
<td>AvgCyclomaticModified</td>
<td>-0.019 (p &lt; 0)</td>
<td>0.051 (p &lt; 0)</td>
<td>0.031 (p &lt; 0)</td>
</tr>
<tr>
<td>AvgCyclomaticStrict</td>
<td>-0.008 (p &lt; 0)</td>
<td>0.074 (p &lt; 0)</td>
<td>0.014 (p &lt; 0)</td>
</tr>
<tr>
<td>AvgEssential</td>
<td>0.033 (p &lt; 0)</td>
<td>0.006 (p &lt; 0)</td>
<td>-0.026 (p &lt; 0)</td>
</tr>
<tr>
<td>AvgLine</td>
<td>-0.039 (p &lt; 0)</td>
<td>0.077 (p &lt; 0)</td>
<td>0.060 (p &lt; 0)</td>
</tr>
<tr>
<td>AvgLineBlank</td>
<td>-0.032 (p &lt; 0)</td>
<td>0.105 (p &lt; 0.1)</td>
<td>0.105 (p &lt; 0.1)</td>
</tr>
<tr>
<td>AvgLineCode</td>
<td>-0.03 (p &lt; 0)</td>
<td>0.045 (p &lt; 0)</td>
<td>0.031 (p &lt; 0.1)</td>
</tr>
<tr>
<td>AvgLineComment</td>
<td>-0.056 (p &lt; 0)</td>
<td>0.127 (p&lt;0.1)</td>
<td>0.058 (p &lt; 0)</td>
</tr>
<tr>
<td>CountDeclClass</td>
<td>-0.058 (p &lt; 0)</td>
<td>0.068 (p &lt; 0)</td>
<td>0.259 (p &lt; 0.001)</td>
</tr>
<tr>
<td>CountDeclFunction</td>
<td>-0.044 (p &lt; 0)</td>
<td>0.137 (p &lt; 0.01)</td>
<td>0.521 (p &lt; 0.001)</td>
</tr>
<tr>
<td>CountLine</td>
<td>0.028 (p &lt; 0)</td>
<td>0.403 (p &lt; 0.001)</td>
<td>0.503 (p &lt; 0.001)</td>
</tr>
<tr>
<td>CountLineBlank</td>
<td>0.048 (p &lt; 0)</td>
<td>0.325 (p &lt; 0.001)</td>
<td>0.509 (p &lt; 0.001)</td>
</tr>
<tr>
<td>CountLineCode</td>
<td>0.026 (p &lt; 0)</td>
<td>0.303 (p &lt; 0.001)</td>
<td>0.487 (p &lt; 0.001)</td>
</tr>
<tr>
<td>CountLineCodeDecl</td>
<td>0.018 (p &lt; 0)</td>
<td>0.331 (p &lt; 0.001)</td>
<td>0.529 (p &lt; 0.001)</td>
</tr>
</tbody>
</table>

machine learning techniques.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value 1 (p &lt; 0)</th>
<th>Value 2 (p&lt;0.0001)</th>
<th>Value 3 (p&lt;0.0001)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountLineCodeExe</td>
<td>0.044</td>
<td>0.369</td>
<td>0.435</td>
</tr>
<tr>
<td>CountLineComment</td>
<td>0.037</td>
<td>0.436</td>
<td>0.411</td>
</tr>
<tr>
<td>CountLineInactive</td>
<td>0.009</td>
<td>0.4</td>
<td>0.330</td>
</tr>
<tr>
<td>CountLinePreprocessor</td>
<td>-0.037</td>
<td>0.150</td>
<td>0.544</td>
</tr>
<tr>
<td>CountSemicolon</td>
<td>0.019</td>
<td>0.307</td>
<td>0.483</td>
</tr>
<tr>
<td>CountStmt</td>
<td>0.021</td>
<td>0.309</td>
<td>0.492</td>
</tr>
<tr>
<td>CountStmtDecl</td>
<td>-0.018</td>
<td>0.235</td>
<td>0.536</td>
</tr>
<tr>
<td>CountStmtEmpty</td>
<td>0.027</td>
<td>0.383</td>
<td>-0.029</td>
</tr>
<tr>
<td>CountStmtExe</td>
<td>0.041</td>
<td>0.33</td>
<td>0.450</td>
</tr>
<tr>
<td>MaxCyclomatic</td>
<td>0.106</td>
<td>0.328</td>
<td>0.270</td>
</tr>
<tr>
<td>MaxCyclomaticModified</td>
<td>0.074</td>
<td>0.282</td>
<td>0.267</td>
</tr>
<tr>
<td>MaxCyclomaticStrict</td>
<td>0.08</td>
<td>0.365</td>
<td>0.279</td>
</tr>
<tr>
<td>MaxEssential</td>
<td>0.098</td>
<td>0.162</td>
<td>0.196</td>
</tr>
<tr>
<td>MaxNesting</td>
<td>-0.08</td>
<td>-0.028</td>
<td>-0.126</td>
</tr>
<tr>
<td>RatioCommentToCode</td>
<td>0.00037</td>
<td>0.203</td>
<td>0.205</td>
</tr>
<tr>
<td>SumCyclomatic</td>
<td>0.03</td>
<td>0.315</td>
<td>0.463</td>
</tr>
<tr>
<td>SumCyclomaticModified</td>
<td>0.023</td>
<td>0.319</td>
<td>0.478</td>
</tr>
<tr>
<td>SumCyclomaticStrict</td>
<td>0.033</td>
<td>0.346</td>
<td>0.458</td>
</tr>
<tr>
<td>SumEssential</td>
<td>0.013</td>
<td>0.225</td>
<td>0.394</td>
</tr>
<tr>
<td>Memory Allocation</td>
<td>-0.1222</td>
<td>0.008</td>
<td>0.267</td>
</tr>
<tr>
<td>Memory Deallocation</td>
<td>0.02</td>
<td>0.110</td>
<td>0.072</td>
</tr>
<tr>
<td>CountClassCoupled</td>
<td>0.21</td>
<td>-0.042</td>
<td>-0.080</td>
</tr>
<tr>
<td>PercentLackOfCohesion</td>
<td>0.017</td>
<td>-0.009</td>
<td>0.033</td>
</tr>
<tr>
<td>CountDeclMethod</td>
<td>-0.021</td>
<td>-0.023</td>
<td>-0.062</td>
</tr>
<tr>
<td>MaxInheritanceTree</td>
<td>-0.021</td>
<td>-0.049</td>
<td>-0.108</td>
</tr>
<tr>
<td>CountClassDerived</td>
<td>0.038</td>
<td>-0.30</td>
<td>-0.058</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions and Future Work

5.1 Conclusions

Security failure identification in the earlier steps can extremely reduce the development costs of large projects. The reproducibility of some security failures is complicated and ignored by testers. Hence, they remain in a system for a long time, while they can cause intense consequences. Understanding the types, characteristics, and locations of residual vulnerabilities in a system can help security experts adopt suitable countermeasures. Consequently, it dramatically decreases the cost and time of development and maintenance phases.

In this thesis, two frameworks for security vulnerability classification and prediction based on the vulnerability reproducibility concept are developed. We adopt the Bohrbug (BB) and Mandelbug (MB) terms proposed by Grottke et al. [21]. We then expand them for security vulnerabilities and introduce two terms: BohrVulnerability (BV) (an easily reproduced vulnerability) and MandelVulnerability (MV) (a hard to reproduce vulnerability).

The first attempt of this research is to classify vulnerabilities according to the
difficulty of the security failure reproduction (the BV and MV categories). The class-
ification framework is able to distinguish between BVs and MVs based on the textual
reports and code fixes. To evaluate the effectiveness of the classification framework,
580 collected vulnerabilities of Mozilla Firefox are analysed, and several machine
learning techniques are employed. The results show that the framework is able to
classify 69% of MVs and 65% of BVs with an accuracy of 67%.

Also, the classification framework is able to collect risky files (i.e., files that have
been changed to fix BVs or MVs). To get more understanding about the nature of BVs
and MVs from the source code level, we investigate the potential linear correlations
between the vulnerability categories and the software metrics and some other runtime
environmental factors related to their reproducibility.

We analyse if files with BVs and MVs can be distinguished based on the metrics.
The linear regression models demonstrate that there is not any significant linear
correlation between the metrics and the categories. Hence, we can conclude that
BV and MVs cannot be distinguished at the code level. Then for the BV and MV
categories, the regression model indicates that most of the metrics are correlated
with BVs and MVs. BVs are more concerned with the size of the program. However,
the number of functions, preprocessors, empty statements, and dynamic memory
allocation has strong correlations with MVs. According to the experiment, the Object-
oriented metrics do not contribute to the vulnerability occurrences.

Due to the complexity of the MV reproduction, an MV-prone prediction model is
developed based on the metrics with strong correlations. Identifying MV locations in
the source code informs the developers in the early stage. Consequently, it decreases
the cost and increases the quality of the software system by focusing testers on the
vulnerable parts. Different statistical machine learning techniques are employed to construct models based on the metrics. The results show that 84% of the MV-prone files can be predicted by the Random Forest algorithm with 75% accuracy.

From the results, we conclude that a simple algorithm (e.g., Decision Tree) can perform as well as a complicated algorithm like Random Forest. Both algorithms perform very well for our case study with insignificant differences. One main reason behind their performance (in comparison with Logistic Regression and Naive Bayes) is that the textual reports can be noisy, and these algorithms are robust against noise in the data set. We conclude the definitions of BVs and MVs as follows:

**BV**: A security failure caused by a BV is always reproducible because its activation and error propagation are not complicated. BVs do not deal with complicated source code. Based on the constructed multiple linear regression model, the number of empty statements, inactive lines, and lines with comments have significant correlations with the existence of BVs. It can imply that large and non-complicated software systems are more vulnerable to BVs in comparison with MVs. When a BV is identified in a system, it is easy to find their locations in the source code to remove them.

**MV**: MVs are much more sophisticated than BVs. Conditions activating MVs are so complicated and vague that make the reproducibility of security failures hard. Sometimes, they do not exhibit consistent behaviour. On average, the fix process of MVs is more complicated than the fix process required for BVs. It involves more numbers of line of code and files. Consequently, it has high entropy. A large number of developers are engaged to fix MVs. The multiple linear regression model identifies software with a large number of functions, preprocessor, dynamic memory allocation, and the number of empty statements is more vulnerable to MVs.
The conclusions confirm that textual reports and code fixes can be used to identify the category of vulnerabilities (BVs or MVs). The vulnerability classification helps the development team design and implement appropriate solutions based on the reproducibility of vulnerabilities. Time to fix and human resources can also be estimated. In addition, the software metrics are helpful to predict vulnerabilities based on their reproducibility at the earlier stage. It helps developers focus their testing techniques on MV-prone files in the software. Consequently, the cost and chance of attacks to the software are decreased.

5.2 Limitations

The proposed classification and prediction frameworks can improve the security of software. However, there exist some limitations in the results and conclusions which can be described as follows:

1. The accuracy of the frameworks, like that of other empirical studies, depends on selected applications and security issue reports. Undoubtedly, the results can be noisy since some of the textual reports are incomplete or inaccurate.

2. The results of this research can give insight about a part of security failures as the analysis is conducted on some important security failures. Furthermore, in most bug tracking systems, some security issues are not discovered or are not reported publicly to decrease the chance of future attacks. Mozilla Firefox, like other big projects, has secret security vulnerabilities. Their information is not published in public since they are not fixed.

3. In the Bugzilla repository, different people (e.g., developers, testers, and end users) are able to report a defect and label it as a security, performance, or another type.
Hence, security vulnerabilities might be classified as non-security defects due to lack of security knowledge.

4. For the vulnerability classification based on reproducibility, more relevant features might exist that we have not noticed. Moreover, the feature definition and extraction mechanisms rely on available data in a system under analyses.

5. The distributions of BVs and MVs depend on the nature of a system under analyses (e.g., the system interaction with hardware and development language). The results are reliable for systems developed in C/C++.

6. Some conceptual features (e.g., activation conditions and impacts) are not explicitly documented in our case study. The proposed vulnerability classification framework is expected to extract features automatically. Therefore, we exclude the features which need manual investigations.

7. Transferring the frameworks (developed for Mozilla Firefox) to other projects that is called cross-project prediction [79] is a challenge. Some features in the classification framework are limited to the textual reports. Hence, the performance of the framework is limited to a project under analyses. It implies that automated classification and prediction frameworks might not be possible. However, some software systems might be good cross-project predictors if they have the same textual report and source code structure [79]. These hypotheses require more analyses and investigations that are not in the scope of this research.

5.3 Future Work

As the future work we are planning to consider more issues:

1. Our approach is applied to one industrial software project (Mozilla Firefox) and
5.3. FUTURE WORK

focuses on the important subset of security issues. We are planning to analyse other large open source software systems (such as Chrome browser, Eclipse, and Apache) to generalize the achievements. However, in this research, we selected Mozilla Firefox as it is widely used for research activities.

2. To build the automatic classification framework, some features (e.g., the impact of security defects on confidentiality, integrity, and availability) are ignored as they require manual efforts. We plan to develop a text-mining approach that applies to the description of vulnerabilities and add these features to the framework.

3. The correlations between the classical software metrics (code complexity, program size, and object-oriented) and the BVs and MVs are investigated to get more understanding about the nature of the categories. There might be some other factors that impact the reproducibility of security failures. We are planning to include more complicated metrics related to the different levels such as design, code, and architecture. As security in different levels are perceived differently.

4. We introduce some metrics related to the internal environment of the system (program interaction, concurrency, runtime error, and memory management) which can represent MVs. We will include more complicated metrics to include those factors. Using more metrics related to the categories can help investigate how internal states of a system can influence the number of MVs.

5. In this research, the static analysis and code features are conducted at the file level. However, some research study defects at the module, component, or function levels. We plan to analyse vulnerabilities at these levels. In this case, we might achieve more interesting results.

6. The features are extracted by using the Python script and C code. To make the
frameworks automated and decrease the human interaction, we plan to add a GUI (Graphic User Interface).
Bibliography


