IDENTIFYING SOFTWARE VULNERABILITIES THROUGH TEXTUAL INFORMATION IN BUG DATABASES

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Dumidu S. Wijayasekara

May 2014

Major Professor: Milos Manic, Ph.D.

AUTHORIZATION TO SUBMIT THESIS

This thesis of Dumidu Wijayasekara, submitted for the degree of Master of Science with a Major in Computer Science and titled “Identifying Software Vulnerabilities Through Textual Information in Bug Databases,” has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: ______________________________________Date_______________
Milos Manic, Ph.D.

Committee
Members: ______________________________________Date_______________
Miles McQueen, M.Sc.

______________________________________Date_______________
Raghunath Kanakala, Ph.D.

Department
Administrator: ______________________________________Date_______________
Gregory Donohoe, Ph.D.

Discipline’s
College Dean: ______________________________________Date_______________
Larry Stauffer, Ph.D.

Final Approval and Acceptance

Dean of the College
of Graduate Studies: ______________________________________Date_______________
Jie Chen, Ph.D.

ABSTRACT

Software vulnerabilities are mistakes in software such that its execution can violate the security policy. Software vulnerabilities are an increasing security focus as critical and sensitive systems become dependent on complex software systems. Therefore, discovering these vulnerabilities as early as possible is of extreme importance. Hidden Impact Bugs (HIBs) are vulnerabilities identified as such, only after the related bug had been publically disclosed. This thesis provides a framework for identifying software vulnerabilities via HIBs using information extracted from publically available bug databases.

The contributions of this thesis are four fold: 1) the concept of HIBs is introduced and the existence of HIBs in software is shown, 2) methodology for identifying software vulnerabilities using textual information from bug databases is presented, 3) information extraction and compression methodologies specific to extracting information from bug databases is provided, 4) a novel methodology for determining the optimal set of dimensions for classification is presented.

ACKNOWLEDGEMENTS

First, I would like to acknowledge and thank my major professor and mentor Prof. Milos Manic for his help, support and guidance. Second, I would like thank Mr. Miles McQueen and Dr. Raghunath Kanakala for agreeing to join my thesis committee and for providing valuable feedback on my work. I would also like to thank Jason Wright, Lawrence R. Wellman, and Jason Larsen for their help and support. Finally, I would like to acknowledge the Idaho National Laboratory for providing financial support for my work.

I also wish to thank Debbie McQueen, Alice Allen and Sara Moore for their assistance and immense support during the academic process.

TABLE OF CONTENTS

Authorization To Submit Thesis ii

Abstract iii

Acknowledgements iv

List Of Figures viii

List Of Tables ix

Chapter 1 Introduction 1

1.1 Organization Of The Thesis 2

1.2 Contributions Of The Thesis 3

Chapter 2 Hidden Impact Bugs 5

2.1 Software Vulnerabilities 5

2.1.1 Methods For Identifying Software Vulnerabilities 6

2.2 Hidden Impact Bugs (HIBs) 7

2.2.1 Definition Of Hidden Impact Bugs (HIBs) 8

2.2.2 Hidden Impact Bugs And Misclassified Bugs In Bug Databases 9

2.3 Bug Databases 10

2.3.1 Bug Reports 12

2.3.2 Bug Life Cycle 13

2.4 Hidden Impact Bugs In Commonly Used Software 14

2.4.1 Selecting HIBs 15

2.4.2 HIBs In The Linux Kernel 16

2.4.3 HIBs In MySQL Server 18

2.5 Software Vulnerability Identification Using HIBs 20

2.5.1 Necessity Of Identifying HIBs In Software 20

2.5.2 Software Vulnerability Identification Using HIBs 22

2.6 Conclusions 23

Chapter 3 Text Mining Bug Databases For Identifying Vulnerabilities 24

3.1 Text Mining Bug Databases 24

3.2 Text Mining For Document Classification 25

3.2.1 Extracting Textual Description 25

3.2.2 Extracting Syntactical Information 26

3.2.3 Compressing Extracted Information 27

3.2.4 Feature Vector Generation 30

3.3 Text Mining Bug Databases For Identifying Vulnerabilities 32

3.4 Conclusion 34

Chapter 4 Classification Of Hidden Impact Bugs 35

4.1 Classification Algorithms 35

4.1.1 Naïve Bayes (NB) And Naïve Bayes Multinomial (NBM) Classifiers 35

4.1.2 Decision Tree (DT) Classifiers 37

4.2 Classification Of HIBs In Bug Databases 40

4.2.1 Classification Subset 40

4.2.2 Construction Of The TDM 41

4.2.3 Test Metrics 42

4.2.4 Bayesian Detection Rate 43

4.2.5 Classification Results 47

4.3 Conclusion 49

Chapter 5 Information Gain Based Dimensionality Selection 51

5.1 Dimensionality Selection Problem In Text Mining 51

5.1.1 Dimensionality Dependent Classification Accuracy 53
5.2 Background 55
5.2.1 Genetic Algorithms (GA) 55
5.2.2 Information Gain (IG) 57
5.3 IG Based Dimensionality Selection For Text Mining Applications 59
5.4 Experimental Results 61
5.5 Conclusion 64
Chapter 6 Conclusion And Future Work 65
6.1 Final Conclusion 65
6.2 Future Work 66
References 68
Appendix A – List Of Publications 75
Journal Publications 76
Peer-Reviewed Conference Publications 76

LIST OF FIGURES

Figure 1. Timeline of Hidden Impact Bugs and impact delay 8
Figure 2. HIBs in bug databases 9
Figure 3. Number of bugs reported to the Redhat Bugzilla bug database 11
Figure 4. Number of bugs reported per day in the Redhat Bugzilla bug database 11
Figure 5. Typical bug report in the Redhat Bugzilla bug database.. 12
Figure 6. Typical bug life cycle of a Bugzilla bug report 14
Figure 7. Number of HIBs by impact delay for Linux Kernel... 17
Figure 8. Number of HIBs that existed per day for the Linux Kernel 17
Figure 9. Number of HIBs by impact delay for MySQL Database Server............................ 19
Figure 10. Number of HIBs that existed per day for the MySQL Database Server 19
Figure 11. Misclassified bug timeline.. 20
Figure 12. Percentage of bugs fixed against time 21
Figure 13. Proposed software vulnerability detection framework using HIBs...................... 22
Figure 14. Text mining bug databases for HIBs architecture 31
Figure 15. Typical binary Decision Tree (DT) classifier.. 38
Figure 16. Pseudocode for C4.5 algorithm 39
Figure 17. Bayesian detection rate for the HIB classification problem 44
Figure 18. Yearly classification results for NB classifier 47
Figure 19. Yearly classification results for NBM classifier 47
Figure 20. Yearly classification results for DT classifier 48
Figure 21. Averaged true positive rate for different numbers of keywords........................... 52
Figure 22. Averaged true negative rate for different numbers of keywords.......................... 53
Figure 23. Averaged classification rate for different numbers of keywords 54
Figure 24. Pseudocode for Genetic Algorithm (GA).. 56
Figure 25. Block diagram of the presented IG based dimensionality selection method......... 59
Figure 26. Averaged true positive rate for each iteration for each method 62
Figure 27. Averaged true negative rate for each iteration for each method 62
Figure 28. Averaged Bayesian detection rate for each iteration for each method 63

LIST OF TABLES

Table 1. Number of bugs reported to the Redhat Bugzilla bug database 10
Table 2. Hidden Impact Bugs (HIBs) in the Linux Kernel 16
Table 3. Hidden Impact Bugs (HIBs) in the MySQL Database Server 18
Table 4. Dimensionality of the bag-of-words after each text mining step............................. 33
Table 5. Number of selected regular bugs and HIBs for classification 41
Table 6. Dimensionality of the bag-of-words after each text mining step............................. 41
Table 7. Confusion matrix for classification of HIBs.. 42
Table 8. Overall classification results 46
Table 9. Number bugs classified as of potential vulnerabilities on a given day in 2011........ 46
Table 10. Averaged classification results for each method. 64

Chapter 1 INTRODUCTION

Software vulnerabilities can be defined as "*an instance of a mistake in the software such that its execution can violate the explicit or implicit security policy*" [Krsul 98], [Ozment 07], [Wright 13]. New software vulnerabilities are discovered in commercial, large scale software every day [Ventor 04]. The actual number of vulnerabilities existing in a software package at a given time is not known and the fact that whether a software package gets more secure over time is still debated [Wright 13]. Furthermore, it has been shown that the number of unidentified vulnerabilities in a given software package might be significantly higher than previously estimated [Wright 13].

Recent trend towards automation and interconnection in infrastructure has lead to critical and sensitive systems which operate critical infrastructure becoming increasingly dependent on complex software systems. Thus, the possibility of software vulnerabilities that threaten the security and integrity of critical infrastructure has sparked an increasing security focus towards identifying software vulnerabilities [Wijayasekara 12]. Discovering these software vulnerabilities as early as possible, at every stage of the software lifecycle, is therefore of extreme importance.

Various methodologies for identifying software vulnerabilities during the software development phase and during operation phases has been suggested [Wijayasekara 12]. However, it has been shown that these methods are not capable of identifying all existing vulnerabilities in software and there is significant room for improvement [Wijayasekara 12], [Austin 11], [Torri 10]. Furthermore, it has been shown that a significant portion of the available tools and methods have a false-positive rate that may overwhelm the identified set of vulnerabilities [Zitser 04].

Therefore, this thesis presents a novel methodology that utilizes textual information in publically available bug databases, to identify software vulnerabilities that have not yet been identified as such. The presented methodology utilizes information extracted from bug reports that have been later identified to be vulnerabilities [Arnold 09]. This phenomenon, where a vulnerability is reported to a bug database as a bug before the full severity of that vulnerability is discovered, is known as Hidden Impact Bugs (HIBs) [Arnold 09],

[Wijayasekara 12]. Thus, relevant information extracted from HIBs can be used to classify bugs as potential vulnerabilities as they are being reported to bug databses.

The remainder of this Chapter describes the organization of this thesis, and then briefly outlines the contributions of this thesis.

1.1 ORGANIZATION OF THE THESIS

Chapter 2 provides an overview of HIBs and bug databases. The Chapter starts by defining the HIBs and other relevant terms related to HIBs and the work presented in this thesis. An overview of typical publically available bug databases in then provided. The Chapter follows with an analysis of HIBs existing in commonly used commercially available software packages. The analysis was performed on 2 commonly used software packages, namely the Linux Kernel and MySQL Database Server. A novel framework for identifying software vulnerabilities by leveraging information extracted from HIBs is then presented in this Chapter. Finally, the Chapter is concluded by reviewing HIBs and the existence of HIBs in commonly used software, and possible implications of HIBs in these software.

Chapter 3, first, introduces text mining for information extraction from textual documents. Each step of the text mining process is then detailed along with their necessity and importance. Text mining of bug databases for extracting various types of information that have been previously done is reviewed next. Next, algorithms and tools used specifically for text mining bug databases in order to identify software vulnerabilities are discussed. The Chapter is concluded by identifying the importance of each text mining step and how it can affect classification accuracy for the problem discussed in this thesis.

Chapter 4 details the classification and evaluation of the software vulnerability identification method that was presented in the previous Chapter. First, details of the different classification algorithms that were used are provided. Then, the specific experimental setup along with evaluation metrics used to compare classifiers are discussed. The problem associated with classification of differently proportioned classes, called the base-rate fallacy is discussed next. The implications of the base-rate fallacy and the Bayesian detection rate of the presented classifiers are also discussed in this Chapter. The experimental results from classification are presented next. Finally, this Chapter is concluded by discussing

the classification results and possible improvements that can be made to further increase the classification accuracy.

Chapter 5 presents a novel Information Gain (IG) based dimensionality selection method for text mining applications. The presented method utilizes the relative Information Gain, and Genetic Algorithms (GA) to extract the optimal set of dimensions for classification. This Chapter first discusses the dimensionality selection problem in text mining applications, and details problems associated to the vulnerability identification problem discussed in this thesis. Then, background information about Genetic Algorithms (GA) and Information Gain (IG) is presented. The novel, Information Gain based dimensionality method is detailed next. The presented method was applied to the problem of text mining bug database for identifying vulnerabilities, and the experimental setup and results are detailed next. Finally, Chapter 5 is concluded by discussing the importance of dimensionality selection and other possible improvements to the presented method.

Finally, Chapter 6 provides overall conclusions and suggests directions for future work.

1.2 Contributions Of The Thesis

The primary contributions of this thesis are four fold.

First, the concept of Hidden Impact Bugs (HIBs) is introduced and an analysis of HIBs in 2 commonly used software packages is performed exposing the existence of HIBs, as well as a trend towards an increase of HIBs.

Second, a novel methodology for identifying software vulnerabilities that are yet to be identified, using textual information extracted from publically available bug databases is presented. The presented method utilizes information in bug reports that were later identified to be vulnerabilities (HIBs).

Third, textual information extraction and compression methodologies specific to extracting information from bug databases is provided. The presented methods identify and extract syntactical information of bug reports in the form of words and compress the extracted information with minimal loss of information and generate feature vectors that can be read by classification algorithms.

Fourth, a novel methodology that utilizes information theory for determining the optimal set of dimensions for text mining based classification is presented. The presented method identifies the optimal set of words for classification using genetic algorithms driven by information gain.

The presented framework and methodologies are tested using bug databases for two commonly used, large scale software distributions, namely the Linux Kernel and the MySQL Database Server.

Chapter 2 HIDDEN IMPACT BUGS

This Chapter first provides a background overview of software vulnerabilities and current methods of detecting software vulnerabilities. Second, Hidden Impact Bugs (HIBs) are introduced, followed by explanation misclassified bugs and impact delay. Next, bug databases and details of bug reports along with the life cycle of bugs are discussed. HIBs in commonly used software are analyzed next, showing the significant presence of HIBs, and a trend towards increasing HIBs. Finally, a framework for identifying software vulnerabilities using information extracted from HIBs is proposed.

2.1 SOFTWARE VULNERABILITIES

Software vulnerabilities can be defined as "*an instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policy*" [Krsul 98], [Ozment 07], [Wright 13]. Thus, all software defects are not vulnerabilities. Further, as per the definition, software vulnerabilities are a subset of software defects also known as software bugs. However, the limiting factor that differentiates software vulnerabilities from software bugs is the security impact of vulnerabilities [Wright 13].

Since it is impossible to guarantee the absence of defects in any software, it is safe to assume vulnerabilities can be present in software that are used in critical and sensitive systems [Shahmehri 12]. Furthermore, many researchers believe that about 5% of all software defects are vulnerabilities [Shahmehri 12], and it has been shown that the actual number of vulnerabilities that exist in software may be 5 to 7 times the number of already known vulnerabilities [Wright 13].

Thus, quick and correct identification of vulnerabilities lead to reducing the time that critical systems are vulnerable to attack. Furthermore, vulnerability discovery enables efficient resource allocation for patch creation as well as threat mitigation and can be used for risk assessment and fault tolerance assessment of systems [Shahmehri 12].

2.1.1 Methods For Identifying Software Vulnerabilities

Software vulnerability discovery is largely focused on source code analysis. The source code based vulnerability discovery methodologies can be divided into two groups: 1) text mining source code, and 2) static code analysis.

Yamaguchi et al. used text mining techniques to extract API symbols and discover usage patterns in source code [Yamaguchi 11]. The extracted information was then converted to a feature vector and supervised machine learning algorithms were used to identify vulnerabilities. The method was tested on 420 functions in the Linux Kernel. However, the classification results were shown to be below expectations [Yamaguchi 11].

Significant portion of previous studies on vulnerability discovery focus on static code analysis and static code analysis tools. However, it has been shown that there are no universal static analysis tools that can provide satisfactory results for vulnerability discovery by itself [Kester 10], [Li 10], [Austin 11]. The existing tools are also very difficult to use because of the large size of software distributions [Khoo 10].

In [Torri 10], Torri et al. evaluated 10 free and open source static code analysis tools on embedded C programs. It was found that while the results were very poor, even the best performing tool needed to be tweaked extensively to produce good results. Therefore, this approach was impractical for use in vulnerability discovery in the software development process [Torri 10]. Similarly, in [Li 10], Li and Cui compared 7 free and open source static analysis tools and concluded that each by itself did not provide a satisfactory discovery of all vulnerabilities. Thus, it was proposed that either a variety of tools be used to compensate for the deficiencies of each tool, or a combination of dynamic and static analysis methods should be used [Li 10]. Kratkiewicz and Lippmann tested 5 different static code analysis tools on 291 small C programs in [Kratkiewicz 05]. It was shown that while some tools were accurate, others were not. Thus it was concluded that it is difficult to select a static analysis tool that will provide good results and more complex test cases may be even difficult to analyze [Kratkiewicz 05]. In [Kester 10] the authors tested 3 static code analysis tools on 12 example programs and showed that the tools does not provide good results and therefore conclude that use of a combination multiple methods is more suited [Kester 10].

Zitser et al. tested five static analysis tools on three open source programs [Zitser 04]. The authors report low detection rates for most of the tools. Furthermore, it was shown that

the best performing tools reported very high false positive rates (false alarm for every 12 to 46 lines of source code) [Zitser 04]. Thus even with high rate of vulnerability identification, the tested static code analysis tools may not provide useful information as the user will be overwhelmed by the large number of false positives.

Li and Leung used machine learning techniques to identify software defects in source code using static code metrics [Li 11].

Other features in the code such as imports, function calls, dependencies between packages have been used for vulnerability prediction [Neuhaus 07], [Neuhaus 09], [Shahmehri 12]. In [Neuhaus 07] the authors discovered that vulnerable components may share similar sets of imports and function calls. Thus, the authors suggest the use of these metrics along with machine learning based classifiers to predict vulnerable components. In [Venter 04] the authors used historical data to predict where the next vulnerability might occur. Various related metrics such as vulnerability density metrics, code-churn, code-complexity and developer activity have been used for vulnerability and fault discovery aw well [Alhazmi 09], [Bell 11]. Dynamic taint analysis has also been proposed to be used for vulnerability discovery in recent years [Zhanga 12].

Austin and Williams showed that no single technique was able to discover every type of vulnerability by itself and therefore, a combination of methods may be the optimal means of vulnerability discovery [Austin 11].

Schumacher et al. showed the importance of gathering information from vulnerability databases to aid the discovery of vulnerabilities in software [Schumacher 00].

Thus, existing methods fail to identify software vulnerabilities with satisfactory accuracy. While researchers suggest various methods, none of them alone can identify all vulnerabilities and there is room for improvement in vulnerability discovery methodologies.

2.2 Hidden Impact Bugs (HIBs)

This section first introduces the concept of Hidden Impact Bugs (HIBs) and impact delay. Next, the implications of HIBs in bug databases are discussed.

2.2.1 Definition Of Hidden Impact Bugs (HIBs)

Hidden Impact Bugs (HIBs) can be defined as those vulnerabilities identified as such only after the related bug had been disclosed to the public [Arnold 09], [Wijayasekara 12]. Thus, these vulnerabilities are disclosed to the public before being identified as having a high security impact and being labeled as vulnerabilities. The disclosure can be via bug reports reported to bug databases or bug patches that are distributed by the software vendor [Wijayasekara 12].

Therefore, in the case of HIBs, even though a software vulnerability is known to the community in the form of a bug, it may not be as quickly fixed by developers, and if a patch is available, it may not be applied in an appropriately timely fashion by end-users, because the security implication of the bug has not been correctly identified [Arnold 09], [Wijayasekara 12].

The time period from which the public disclosure occurs to the identification of the true security impact of the HIB is called the impact delay [Arnold 09], [Wijayasekara 12]. The timeline of a HIB and the impact delay is depicted in Figure 1. This time period is crucial since during this time systems that use the software package are vulnerable to outside threats. Because the bug that causes the vulnerability has already been identified, this time can be reduced if the security implications of the bug are identified as it is reported to the bug database. Furthermore, intelligent adversaries may identify the security implications by monitoring bug databases and use the information to attack critical and sensitive systems during the impact delay.

Figure 1. Timeline of Hidden Impact Bugs and impact delay

2.2.2 Hidden Impact Bugs And Misclassified Bugs In Bug Databases

As mentioned in the previous section, some vulnerabilities are reported as bugs to bug databases, and are not identified vulnerabilities till later. Figure 2 shows a Venn diagram of software bugs, in terms of vulnerabilities, bug reports, and HIBs. While the proportions of the figure might not be accurate, it clearly depicts the problem domain discussed in this thesis.

According to the definition of HIBs, HIBs were reported as bug and later identified as vulnerabilities. Thus, HIBs reside in the intersection between identified vulnerabilities and bug reports.

However, the intersection between all the vulnerabilities of the software and bug reports is much larger. This is because some vulnerabilities that have not yet been identified reside in the bug database as bug reports. These bugs have been misclassified as typical bugs and their true security impact has not yet been identified. Furthermore, it has been shown that the number of misclassified bugs has been significantly underestimated and is much larger than previously thought [Wright 13].

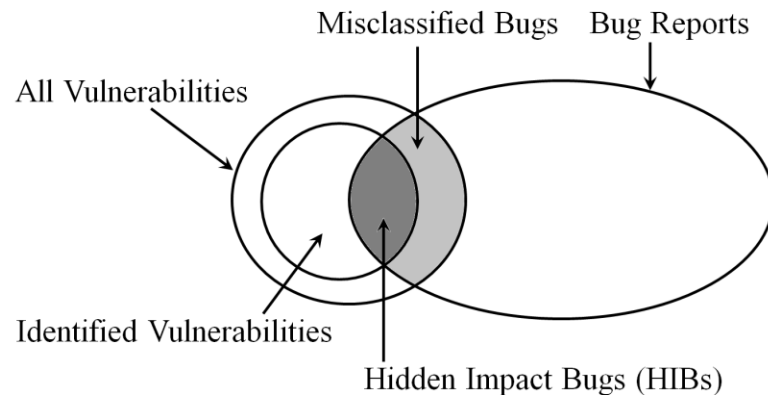

Figure 2. HIBs in bug databases

Therefore, it might be possible to leverage the information that is stored in bug databases as HIBs, to automatically identify the set of misclassified bugs. Furthermore, this information can also be used to identify the true security impact of a bug while it is being reported to the bug database, thus reducing the impact delay and the time period that critical systems are vulnerable to attacks [Wijayasekara 12].

Table 1. Number of bugs reported to the Redhat Bugzilla bug database

Year	Number of bug reports	Number of bug reports per day
From Nov. 1998	336	5.5
1999	3,788	10.4
2000	5,846	16
2001	7,839	21.5
2002	9,200	25.3
2003	8,497	23.3
2004	11,951	32.7
2005	12,428	34
2006	15,249	41.8
2007	17,217	47.2
2008	20,817	57.0
2009	26,950	73.8
2010	43,120	118.1
To April 2011	17,616	146.8
Unknown	2108	-
Total	**202,896**	**44.3**

2.3 BUG DATABASES

Bug databases for software are kept in order to keep track of the bugs existing in the software and identifying which bugs are patched. Publically available bug databases benefit from information provided by typical software users with a diverse set of technical backgrounds as well as programmers and developers [Noll 11]. These bug databases enable developers to identify previously unidentified bugs in the software and at the same time users can track the resolution process of each bug.

It has been shown that these bug databases are extremely useful in increasing the quality and reliability of software as well as containing vital information that can be used for various purposes such as improving future design requirements [Ko 06], gathering vital feedback from users [Noll 11], and improving software reliability [Ahmed 08], [Ahmed 09].

In this section, bug reports from the Redhat Bugzilla bug database are analyzed [Redhat 14]. The Redhat Bugzilla bug database was selected because: 1) it is one of the most extensive bug databases available, 2) all other Bugzilla bug databases generally follow the same format, and 3) most of the Linux vulnerabilities that is examined in this thesis are related to bugs in the Redhat Bugzilla database. Although the Redhat Bugzilla database "*is not an avenue for technical assistance or support, but simply a bug tracking system*" [Redhat 14], it has been shown that certain details in the bug reports can be used for various forms of classification as mentioned in Section 3.1 [Lamkanfi 10], [Lamkanfi 11], [Ko 06].

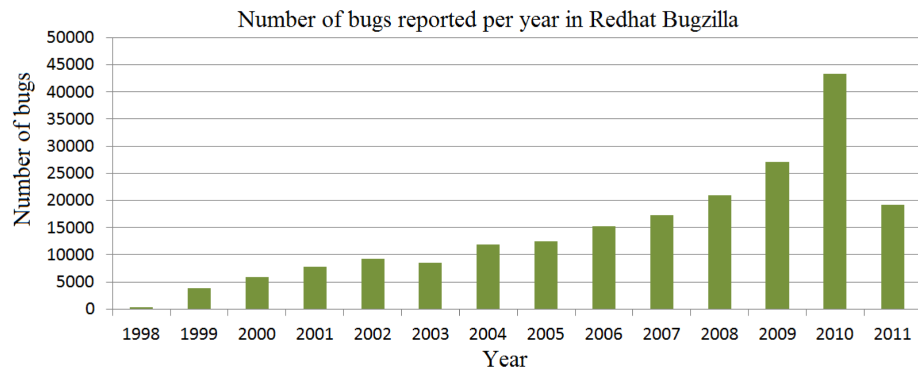

Figure 3. Number of bugs reported to the Redhat Bugzilla bug database

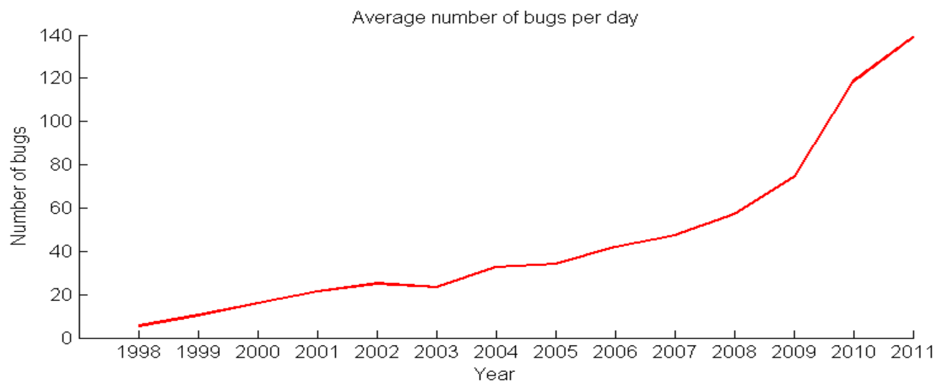

Figure 4. Number of bugs reported per day in the Redhat Bugzilla bug database

As of April 2011, the Redhat Bugzilla bug database contained 202,896 entries. The first bug, which is a test bug report, was added to the database on the 1st of November 1998. Table 1 shows the distribution of bugs per year and the mean number of bugs per day in the Redhat Bugzilla bug database. As shown by Figure 3 and Figure 4, the number of bugs reported as well as the number of bugs reported per day has been steadily increasing through the years. This might be due to the fact that mature releases of the same software tend to have more bugs reported [Ahmed 09]. A similar overview of the MySQL database server bug reports can be found in [Wright 13].

Figure 5. Typical bug report in the Redhat Bugzilla bug database

2.3.1 Bug Reports

A bug may be reported to a bug database by a typical user of the software package, a software developer, or an automated bug reporter. Before reporting, it is the responsibility of the bug reporter to search whether the bug has occurred before and reported to a bug database. Furthermore, the reporter has to provide a clear description of the bug, so that the development team is able to patch it within an appropriate time frame.

Once a bug is reported to the bug database, it is first reviewed and the reported bug is assigned a bug ID, which is a unique identifier. The bug is then entered in to the bug resolution process (Section 2.3.2 details the typical life cycle of a bug).

Figure 5 shows a typical bug report that was reported to the Redhat Bugzilla bug database. When a bug is reported, the reporter can assign different parameters to the bug report as he or she sees fit. These parameters include terms such as severity, priority, product, component and keywords. During the life cycle of the bug, these parameters may be changed according to its nature and severity, and increased understanding of the impact of the bug.

Apart from these set parameters, the person who reports the bug must provide a title for the bug which is known as the short description of the bug. As the name suggests it is a short description of the bug that gives an overall understanding of the bug. A long description of the bug is also provided by the user that should describe the bug in more detail. Depending on expertise and the requirements of the bug reporter the long description may include code snippets, how to recreate the bug, how often can the bug be recreated, the specifications of the hardware setup etc., which are meant to enable the developer to more easily identify and rectify the bug. Furthermore, in some cases, especially for automated bug reports, a memory dump is also attached to the long description of the bug report.

Comments can also be added by users and administrators to convey the progress and development of the bug fix or other relevant facts.

2.3.2 Bug Life Cycle

A bug is assigned a status which describes the current position of the bug in the bug resolution process. The status of the bug changes according to the position of the bug in the life cycle, thus allowing users and the development team to be informed on the progress of the bug fix.

Once a bug is reported, the development team needs to identify whether the bug is actually a bug and it has not been reported before. Since some bug reports are actually feature requests and some faults might not be bugs in the software [Wright 13], initially a bug is assigned an “unconfirmed” tag.

If the reported bug is confirmed as a bug, then it is entered into the bug resolution process. During this process the development team has to correctly identify the severity of the bug in order to prioritize it among the set of bugs to be fixed. Although the bug reporter initially assigns a severity and priority for a bug, this has been shown to be extremely subjective and incorrect at times, and therefore, cannot be relied upon [Arnold 09], [Lamkanfi 10].

Once the severity has been identified, the correct person that is responsible for the bug is identified and assigned to the bug. That person then either resolves the bug or redirects the bug to a different person if the assignment was incorrect.

The basic bug life cycle of a reported bug is shown in Figure 6.

Thus, each of the steps in the bug resolution process increases the time it takes to fix a bug. Furthermore, if the true security impact of the bug is identified as the bug was reported to the bug database, it is safe to assume that the time taken to resolve the bug will be significantly reduced [Wijayasekara 12].

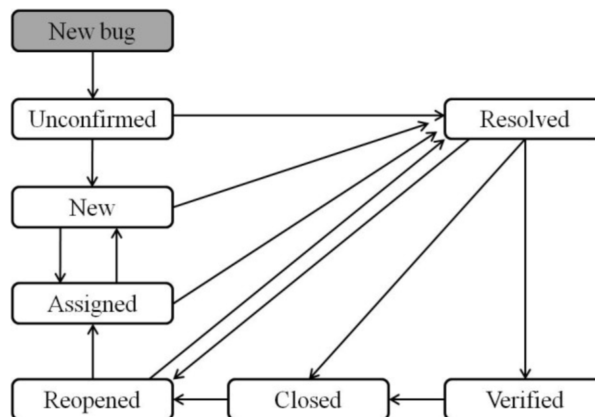

Figure 6. Typical bug life cycle of a Bugzilla bug report

2.4 HIDDEN IMPACT BUGS IN COMMONLY USED SOFTWARE

HIBs in commonly used software packages are identified and analyzed in this section. The software packages tested were Linux Kernel and MySQL Database Server. These software were chosen for: 1) their wide spread usage, 2) public availability of bug databases,

3) large number of reported bugs, 4) public availability of vulnerability databases, and 5) large number of reported vulnerabilities.

Both Linux Kernel and MySQL server have been widely used as they are distributed as free and open source software. Furthermore, many other software use these software packages as their components or as their base, thus increasing the potential corpus of affected software for a given vulnerability.

2.4.1 SELECTING HIBS

The MITRE Common Vulnerabilities and Exposures (CVE) vulnerability database [MITRE 14] was chosen as the vulnerability database for both Linux Kernel and MySQL Database Server. The MITRE CVE database was chosen for its: 1) ease of access, 2) the larger number of vulnerabilities reported, and 3) clear information about the reported date and bugs associated with a vulnerability.

For each software package analyzed, vulnerabilities were divided into two groups depending on when they were first reported: 1) time period from the 1st of January 2006 to the 31st of December 2008, which will be hereafter referred to as the *first time period* and 2) the time period from the 1st of January 2009 to the 30th of April 2011 which will be hereafter referred to as the *second time period*. The first time period corresponds to the time period studied by Arnold et al. in [Arnold 09]. The second time period is the time from the end of the first time period to the time the dataset was extracted from the bug and vulnerability databases for this study.

For analyzing the number of HIBs existing in software, a conservative approach was taken when selecting the vulnerabilities. Thus, specific rules were applied to the vulnerability database selecting only the vulnerabilities that affected: 1) multiple processors, 2) multiple distributions and 3) vulnerabilities that affected a certain version of the software package and above, were selected from the vulnerability database for the two time periods. Vulnerabilities that affected only a single processor were excluded because these vulnerabilities affect only a small subset of users and it is difficult to clarify whether they were caused by necessarily a software issue. Similarly, vulnerabilities that affected only one distribution were excluded

because of their low impact. A version cut-off was used to eliminate counting older vulnerabilities that might not be relevant to current software version.

The public disclosure time for HIB identification was the time a patch was released for the bug related to the vulnerability. Furthermore, only vulnerabilities with an impact delay of at least 2 weeks were selected as HIBs.

2.4.2 HIBs In The Linux Kernel

Table 2 shows the HIBs and the vulnerabilities identified for the Linux Kernel using the above mentioned metrics. A combined total of 403 vulnerabilities were identified for the Linux Kernel in the two time periods. Out of these vulnerabilities, 129 (32%) were HIBs. Furthermore, nearly 15% of the vulnerabilities were HIBs with at least 8 weeks impact delay, meaning that the true impacts of these bugs were only identified 2 months after their public disclosure.

The total number of vulnerabilities in the second time period was 185, which is a 15% reduction from the first time period. However, the number of vulnerabilities with at least 2 weeks of impact delay increased from 56 (25%) to 73 (39%).

Thus, while nearly a third of the vulnerabilities reported from 2006 to 2011 were HIBs, the number and the percentage of HIBs has increased from the first time period to the second time period.

Table 2. Hidden Impact Bugs (HIBs) in the Linux Kernel

Type	Impact Delay	2006-01-01. to 2008-12-31 (*First Time Period*)	2009-01-01. to 2011-04-30 (*Second Time Period*)	Total
Hidden Impact Bugs	$\geq 2\ weeks$	56 (25.69%)	73 (39.46%)	129 (32.01%)
	$\geq 4\ weeks$	38 (17.43%)	55 (29.73%)	93 (23.08%)
	$\geq 8\ weeks$	31 (14.22%)	29 (15.68%)	60 (14.99%)
All Vulnerabilities	**-**	**218**	**185**	**403**

Figure 7 shows the histogram of HIBs in the Linux Kernel in terms of impact delay in weeks. It can be seen that most of the HIBs were correctly classified within 20-30 weeks after their public disclosure. Furthermore, Figure 8 shows the number of HIBs that existed for each day in the Linux Kernel for the given time period. Thus, on average there were nearly 10 HIBs in the Linux Kernel for a given day. The trailing edge of Figure 8 at the end is because new HIBs that are reported to the bug database is not known as of yet.

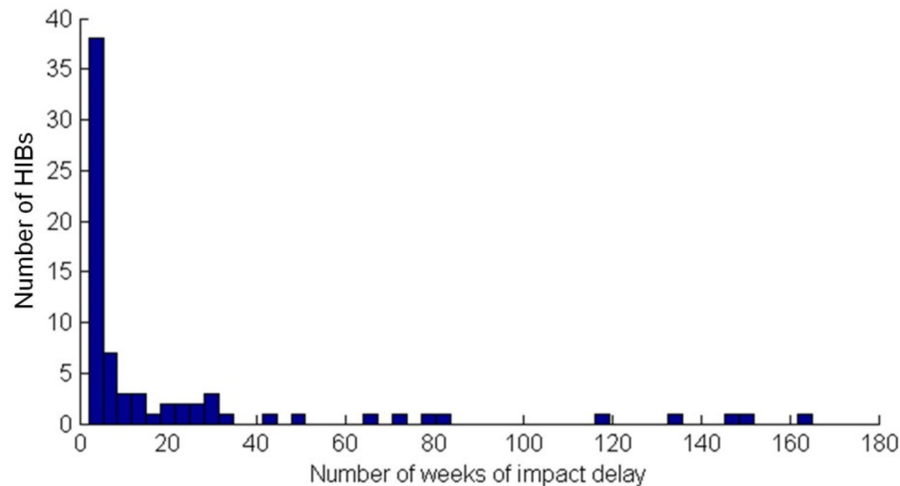

Figure 7. Number of HIBs by impact delay for Linux Kernel

Figure 8. Number of HIBs that existed per day for the Linux Kernel

2.4.3 HIBs In MySQL Server

Similar to Linux Kernel HIBs, Table 3 shows the HIBs and the vulnerabilities identified for the MySQL Database Server. A combined total of 66 vulnerabilities were identified for the MySQL Database Server using the above mentioned rules. Out of these vulnerabilities, over 62% (41) were HIBs with an impact delay of at least 2 weeks. Furthermore, while the number of HIBs in the second time period was reduced, due to the smaller number of total vulnerabilities reported, the percentage of the HIBs was increased in the second time period.

Similar to the Linux Kernel, a significant portion (over half) of all the vulnerabilities reported from 2006 to 2011 were HIBs. The percentage of HIBs has increased from the first time period to the second time period in MySQL Database Server as well.

Furthermore, half of the vulnerabilities were HIBs with more than 8 weeks impact delay. Thus, a significant number of vulnerabilities were identified only after more than 2 months after their public disclosure. This is also reflected in Figure 9 which shows the histogram of HIBs for the MySQL Database Server.

Table 3. Hidden Impact Bugs (HIBs) in the MySQL Database Server

Type	Impact Delay	2006-01-01. to 2008-12-31 (*First Time Period*)	2009-01-01. to 2011-04-30 (*Second Time Period*)	Total
Hidden Impact Bugs	$\geq 2\ weeks$	22 (59.46%)	19 (65.52%)	41 (62.12%)
	$\geq 4\ weeks$	21 (56.76%)	19 (65.52%)	40 (60.62%)
	$\geq 8\ weeks$	17 (45.95%)	16 (55.17%)	33 (50%)
All Vulnerabilities	**-**	**37**	**29**	**66**

Finally, Figure 10 shows the number of HIBs that existed for each day in the MySQL Database Server for the given time period. Thus, on average there were nearly 4 HIBs existing in the MySQL Database Server for a given day.

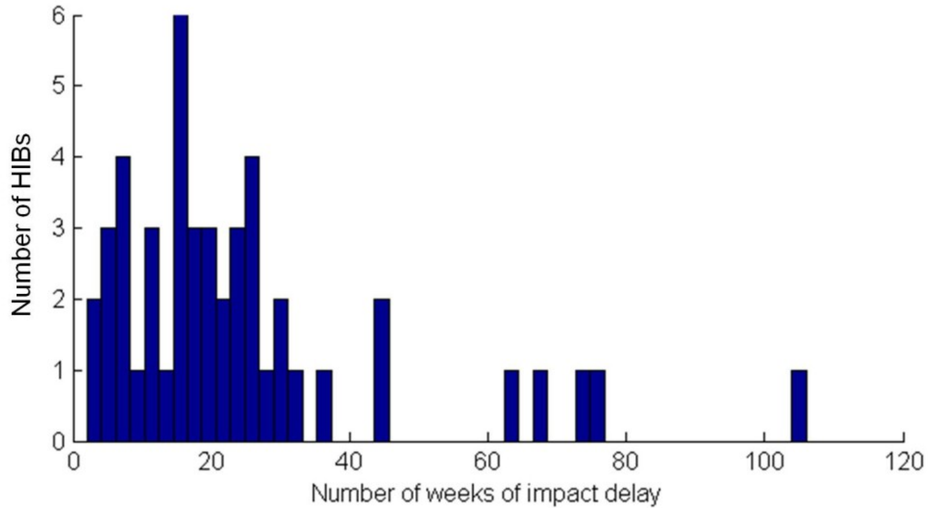

Figure 9. Number of HIBs by impact delay for MySQL Database Server

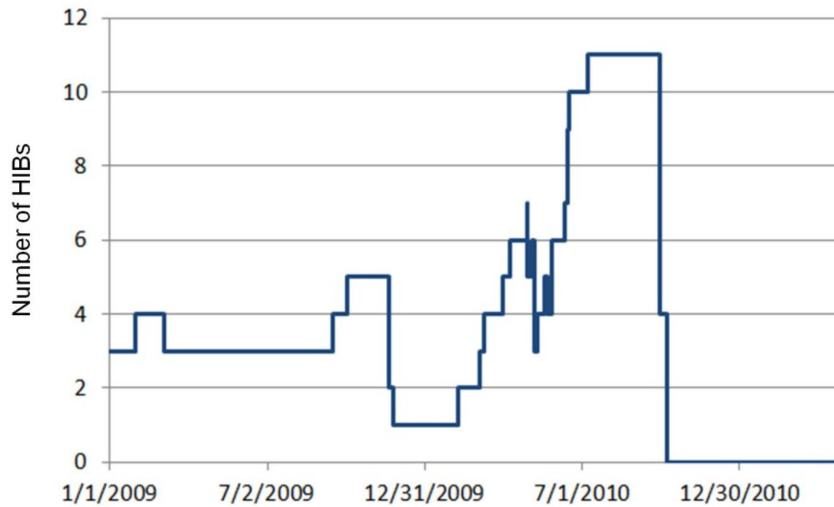

Figure 10. Number of HIBs that existed per day for the MySQL Database Server

2.5 SOFTWARE VULNERABILITY IDENTIFICATION USING HIBS

As shown in the previous section, HIBs are a significant portion of all vulnerabilities. Furthermore, the percentage of HIBs has increased in recent years. Thus, by identifying HIBs as they are being reported to bug databases, and correctly identifying the set of misclassified bugs, a significant portion of the software vulnerabilities can be identified.

This section first illustrates the necessity of identifying HIBs, and then proposes a methodology for identifying software vulnerabilities by using HIBs.

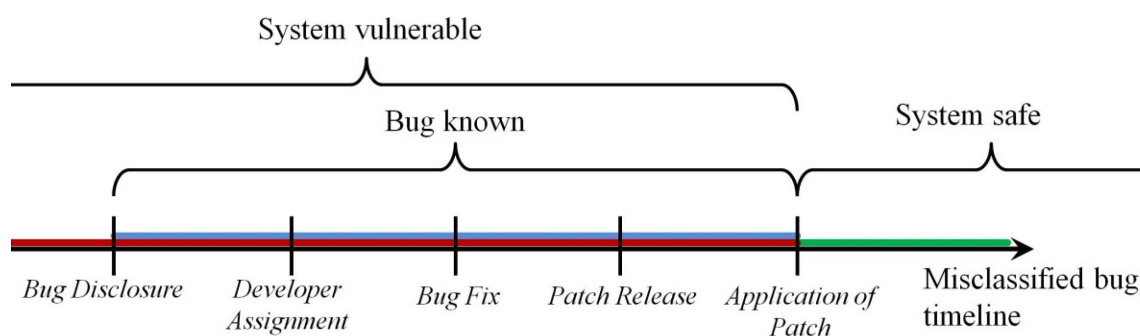

Figure 11. Misclassified bug timeline

2.5.1 NECESSITY OF IDENTIFYING HIBS IN SOFTWARE

The primary necessity of identifying HIBs is to reduce the time systems are vulnerable to security attacks. Figure 11 shows the timeline of a misclassified bug, meaning the bug is actually a vulnerability but has not yet been identified as one. If the true security impact of the bug was known, the time taken for each step of the process might be reduced.

For the HIBs identified for the Linux kernel, it was calculated that the average time taken to create patch was 575 days, while the average time to create a patch for a vulnerability that was identified as such immediately was 387 days. Furthermore, Figure 12 shows the percentage of HIBs and vulnerabilities fixed against the time taken to create a patch. It was identified that nearly 50% of vulnerabilities were fixed within the first 100 days while only less than 30% of the HIBs are fixed.

Software vendors tend to release software patches either when a certain number of fixes accumulate or on a certain day of the week/month. This is primarily because, it is easier for the developers and users to distribute or apply multiple patches simultaneously. However, if the security implication of a bug is high and it is known, then it is likely that the patch will be released as soon as it is created.

Furthermore, since patch application takes time and computing resources, many users and systems administrators tend not to apply patches as they become available if the path does not affect the security of the system [Arnold 09], [Wijayasekara 12]. Again, if the true security impact of the high impact bug is known this time might be reduced.

Finally, because the bug is known to the public from the time of disclosure, if an intelligent adversary identifies the true security impact of a misclassified bug, that information might be leveraged to attack vulnerable systems [Arnold 09], [Wijayasekara 12]. Thus, vulnerabilities that are misclassified as bugs may be more dangerous than other vulnerabilities, because the information about the vulnerability is already disclosed to the public.

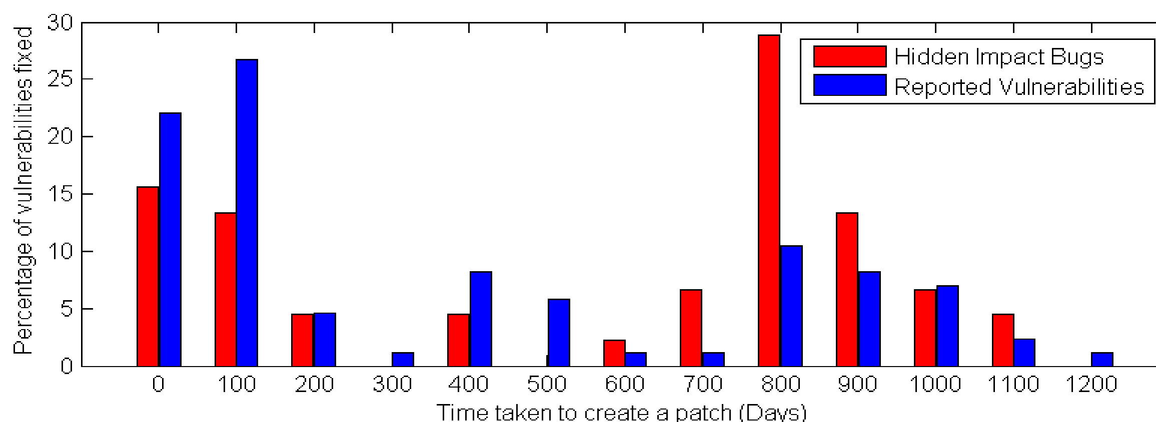

Figure 12. Percentage of bugs fixed against time

2.5.2 Software Vulnerability Identification Using HIBs

Thus, because of the factors elaborated above, identifying the true security impact of a bug while it is being reported to a bug database is important. This section proposes a methodology that utilizes information in HIBs to not only correctly classify new bugs, but also identify misclassified bugs that may be in the bug database.

Figure 13 depicts the proposed software vulnerability detection framework. The proposed framework utilizes automatically extracted knowledge from already known HIBs to train classification algorithms. These classification algorithms will then be able to correctly classify newly reported bugs as potential vulnerabilities or normal bugs. Furthermore, existing bugs can also be classified using the trained classifier to detect misclassified bugs already in the bug database.

Finally, once a classified potential vulnerability is verified by the development team, that information can be used to train the classifier to further increase the classification accuracy.

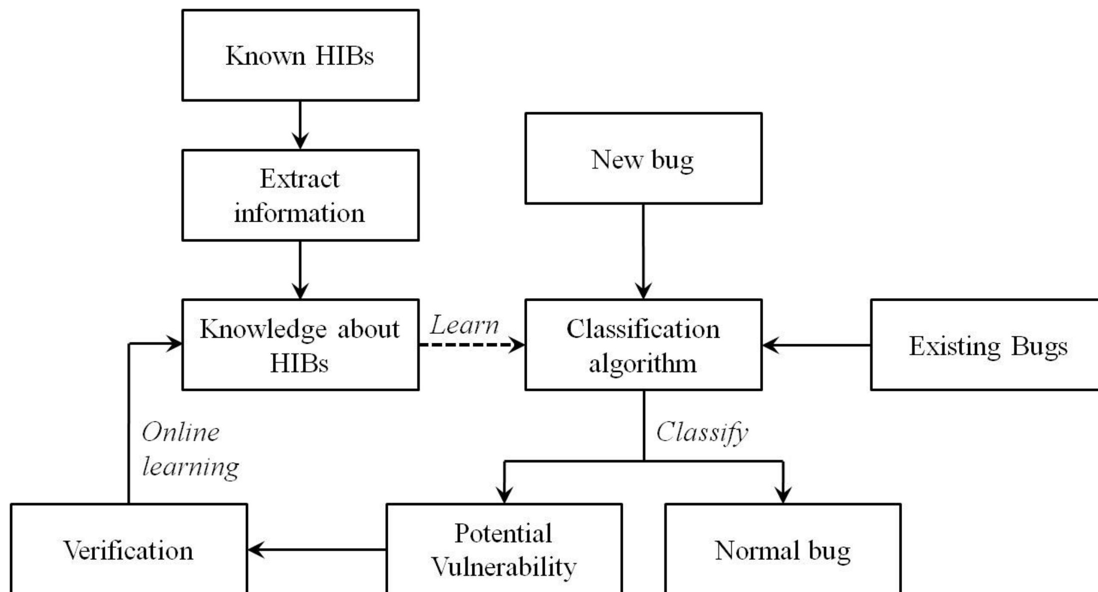

Figure 13. Proposed software vulnerability detection framework using HIBs

2.6 CONCLUSIONS

This Chapter first detailed software vulnerabilities and current vulnerability detection methods. The concept of HIBs was introduced next. An analysis of HIBs in the Linux Kernel and the MySQL Database Server software packages showed the significant presence of HIBs and their increase in recent years. Finally a framework was presented that utilizes information in HIBs to correctly classify bugs as they are being reported to bug databases, as well as correctly classifying misclassified bugs.

Chapter 3 TEXT MINING BUG DATABASES FOR IDENTIFYING VULNERABILITIES

Advanced text mining techniques have been shown to be able to extract information from textual description of bug reports for various classification and clustering purposes [Runeson 07], [Lamkanfi 11], [Wijayasekara 12]. This Chapter first details some previous work done on text mining bug databases for bug the triaging process. Advanced text mining techniques that are capable of extracting relevant information and converting the extracted information into machine readable format is then detailed. Then, a novel text mining framework for extracting textual information from bug reports for software vulnerability identification using information in Hidden Impact Bugs is presented before concluding the chapter.

3.1 TEXT MINING BUG DATABASES

Previous studies have shown that the textual data contained in bug reports may carry important information that can help developers in the bug triaging process. Previous work on bug database mining focuses on three main problems: 1) assigning the correct person to fix a bug, 2) finding duplicate bug reports and 3) assigning the correct severity to a reported bug.

In [Cubranic 04], [Anvik 06] and [Jeong 09], the authors used text mining to assign the correct person to fix a bug. The correct person can be a developer whose expertise is in that area, or a developer who is responsible for the affected code. In [Cubranic 04], Cubranic and Murphy used Naive Bayes to classify bugs contained in the Eclipse bug database. Anvik et al. used a number of classification techniques to classify bugs in the Eclipse and Firefox databases [Anvik 06]. In [Jeong 09], Jeong et al. used a Markov model for the Eclipse and Firefox databases and showed better classification accuracy.

Detection of duplicate bug reports is explored in [Runeson 07], [Wang 08], [Prifti 11] and [Wu 11]. Runeson et al. used vector space and cosine similarity measures to find redundant bugs in a Sony Ericsson mobile bug database [Runeson 07]. In [Wang 08], Wang et al. used similarity measures to detect potential duplicate bugs for Eclipse and Firefox bug

databases. Wu et al. [Wu 11] also proposed a tool for detection of duplicate bugs in Apache, Eclipse and Linux bug databases.

In [Lamkanfi 10] and [Lamkanfi 11] Lamkanfi et al. used the textual description of bug reports to classify severity of bugs. Eclipse, GNOME and Mozilla bug reports were classified into three classes of severity in [Lamkanfi 10] by using a Naïve Bayes classifier. In [Lamkanfi 11] different classification algorithms such as Naïve Bayes and Naïve Bayes Multinomial were compared for classifying Eclipse and GNOME bug reports.

Linux kernel and MySQL DBMS bug reports were used to generate complexity metrics in [Cotroneo 12]. This information was then used to train classifiers such as Naïve Bayes and Decision Trees to predict ageing related bugs in these software.

3.2 Text Mining For Document Classification

The purpose of text mining is to extract relevant information and knowledge from textual data in order to perform a task [Ingersoll 13]. One of the primary tasks in text mining applications is automated classification and clustering of textual documents [Ingersoll 13]. In order to extract knowledge and information from human written textual documents in a machine understandable format, that is computationally efficient and can produce reasonable classification results, several advanced text mining techniques are used. These text mining techniques can be combined and used to extract only relevant information while removing data that yields little to no information [Ingersoll 13].

The text mining process can be separated into 4 steps: 1) extracting textual description, 2) extracting syntactical information, 3) compression of extracted information, and 4) generating a machine readable feature vector [Wijayasekara 13].

3.2.1 Extracting Textual Description

In this step, the relevant portion of the text that is required for the classification is extracted from the document. This is important as information that may not be related to the classification problem may reside in the document, and this information may lead to sub-optimal classification [Ingersoll 13].

Furthermore, all the formatting within the document is removed in this step, as it carries no relevant information [Ingersoll 13].

3.2.2 Extracting Syntactical Information

In this step, syntactical information contained within the document is extracted. While semantical information is ignored it has been shown that extracting and representing semantical information is extremely difficult, highly domain specific, and highly biased towards the writer of the document [Ingersoll 13]. Thus, utilizing semantical information for many cases is difficult. Furthermore, using only syntactical information has been shown to yield good results for text mining applications related bug databases [Lamkanfi 11].

In this step, first the all the unique words are extracted from the document and stored. This is known as tokenizing the document.

Second, all characters are converted to lower case, and numbers and special characters are removed. Since the case of the words carries very little information all words are converted to lower case for easy manipulation [Ingersoll 13]. Furthermore, numbers and special characters has no meaning once taken out of context, therefore, tokenized numbers and special characters carry no relevant information and thus are removed [Ingersoll 13].

Finally, frequently occurring words in the English language, known as *stop words*, are also removed from the tokenized set of words. These words include Pronouns such as: "*I, he, she*", Articles such as: "*a, an, the*", Prepositions such as: "*after, to, but*", Conjunctions such as: "*and, but, when*", and other frequently appearing words. Such words carry very little to no information when taken out of context and occur too frequently to enable distinguishability of documents, and are therefore disregarded. The remaining set of unique words are known as *keywords*.

Thus, a set of documents D containing N documents:

$$D = \{d_1, d_2, ..., d_N\} \tag{3.1}$$

where, d_j is a document, and for the set of D documents, M unique keywords exists. Therefore the set of documents can be represented as:

$$D = \{W_1, W_2, ..., W_M\} \quad (3.2)$$

where, W_i are unique keywords that has been extracted. This representation is known as the "*bag-of-words*" representation. Each document in this representation can be viewed as a set of unique keywords:

$$d_j = \{W_1, W_2, ..., W_m\} \quad (3.3)$$

where, W_i are unique keywords that exists in document d_j and $m \in M$.

3.2.3 Compressing Extracted Information

The main problem faced when using the bag-of-words representation is, as the number of documents N increase, the number of unique keywords M also increase. This results in a large matrix which leads to increased resource usage and higher computational times. Since many of the extracted keywords might not appear in most of the documents, many keywords will not contribute information relevant for classification. Furthermore, many of the keywords can be syntactically unique but semantically identical, meaning while the word is unique, the meaning is the same.

Therefore, once the syntactical information is extracted, the information needs to be compressed. In order to minimize information loss during the compression stage, several text mining techniques can be used.

One such technique is identifying and combining synonyms. Synonyms are words that have the same meaning or nearly the same meaning as another word, i.e. syntactically different but semantically similar. Thus identifying and combining synonyms leads to a reduced dimensionality with very little loss of information. This is typically, done by using English word databases such as Wordnet [Fellbaum 98].

A keyword with r synonyms can be represented as a set of keywords that are its synonyms:

$$\widehat{W}_i = \{W_{i1}, W_{i2}, \ldots., W_{ir}\} \tag{3.4}$$

where, $\widehat{W}_i$ is the set of synonyms for keyword W_i and $r > 0$. Once all possible synonyms for each keyword are identified the bag-of-words representation can be extended so that each keyword is now a set of synonyms:

$$D = \{\widehat{W}_1, \widehat{W}_2, \ldots., \widehat{W}_M\} \tag{3.5}$$

Using the identified synonyms the keywords can be combined. Thus, for two keywords W_i and W_j, with A and B number of synonyms:

$$\widehat{W}_i = \{W_{i1}, W_{i2}, \ldots., W_{iA}\} \tag{3.6}$$

$$\widehat{W}_j = \{W_{j1}, W_{j2}, \ldots., W_{jB}\} \tag{3.7}$$

$$\widehat{W}_i \equiv \widehat{W}_j \;\; if\; any\, W_{ia} = W_{jb} \;\; \forall a \in A, \;\; and \;\; \forall b \in B \tag{3.8}$$

where, $\widehat{W}_i \equiv \widehat{W}_j = \widehat{W}_i \cup \widehat{W}_j = \widehat{W}_{ij}$

$$\widehat{W}_{ij} = \{\widehat{W}_i, \widehat{W}_j\} = \{W_{i1}, W_{i2}, \ldots., W_{iA}, W_{j1}, W_{j2}, \ldots., W_{jB}\} \tag{3.9}$$

then, $\widehat{W}_i$ and $\widehat{W}_j$ are removed from the bag-of-words representation and $\widehat{W}_{ij}$ is added. This process is repeated until equation (3.8) is no longer satisfied for all synonym sets in the bag-of-words.

Once this step is completed, the number of keyword sets in the bag-of-words representation is reduced to M':

$$M' = M - syn \tag{3.10}$$

where,

$$syn = \sum_{i=1}^{M} (R_i - 1) \tag{3.11}$$

where, R_i is the number of keyword sets in $\widehat{W}_i$, of the bag-of-words resulting after all synonyms are combined.

Another technique for compressing information is deconstructing words into their base forms and combining similar words. The deconstruction of words into their base form is known as stemming. Stemming is capable of deconstructing words that have been transformed, for example by pluralizing or by adding a gerund, into their basic form. This enables identification of transformed words as similar to their base words.

The process of indentifying similar stemmed words and combining them is similar to the process described above for synonyms. Therefore as with identifying and combining synonyms, the dimensionality of the bag-of-words is reduced with minimal loss of information.

Identifying the most frequently used keywords or keyword sets in the bag-of-word representation leads to further compression of information. This is done by counting the number of documents d_j that each keyword or keyword set $\widehat{W}_i$ appears in, and selecting the most recurring keyword set. Typically keyword sets that appear in less than $T\%$ of the documents are discarded. This type of threshold selection reduces the dimensionality significantly and identifies words that are most general to the document set. While this method removes many keywords that appear in only a small subset of documents, and therefore cannot contribute to classification, it may also remove words that are important to classification and retain words that may not contribute to, or adversely affect classification.

3.2.4 Feature Vector Generation

Once information compression is done, the textual information can be converted in to a vector of real values, which can be used as an input to a classification algorithm. This is done by representing each document using the number occurrences of each keyword set within the bag-of-words resulting from the information compression step.

Therefore, a document d_j can be represented as:

$$d_j = \{w_{j,1}, w_{j,2}, ..., w_{j,M'}\} \tag{3.12}$$

where, $w_{j,i}$ is the number of times the keyword set $\widehat{W}_i$ occurs in document d_j. M' is the number of keyword sets in the bag-of-words after information compression phase.

Thus a set of documents D containing N documents and M' keyword sets can be represented as a $N \times M'$ matrix. This matrix is known as the Term-Document Matrix (TDM):

$$TDM = \begin{bmatrix} d_1 = \{w_{1,1}, w_{1,2}, ..., w_{1,M'}\} \\ \cdots \\ d_N = \{w_{N,1}, w_{N,2}, ..., w_{N,M'}\} \end{bmatrix} \tag{3.13}$$

This TDM can now be used as an input to a classification algorithm. However, it has been shown that further improvements can be made to classification accuracy by weighing the keyword sets according to their importance [Ingersoll 13]. This is typically done by using the Term Frequency-Inverse Document Frequency (TF-IDF) method [Ingersoll 13].

The TF-IDF method assumes that the importance of a keyword set in a document is inversely proportional to the frequency that the keyword set occurs in all documents. The weight for the keyword set i in bug report j ($\tau_{j,i}$), can be calculated as:

$$\tau_{j,i} = w_{i,j} \times \log\left(\frac{N}{df_i}\right) \tag{3.14}$$

where, $w_{j,i}$ is the number of times the keyword set $\widehat{W}_i$ occurs in document d_j, and df_i is the number of times the keyword set $\widehat{W}_i$ occurs in all N documents. Once, $\tau_{j,i}$ is calculated for all N documents and M' keyword sets, each $w_{j,i}$ is multiplied by $\tau_{j,i}$ to produce the final TDM.

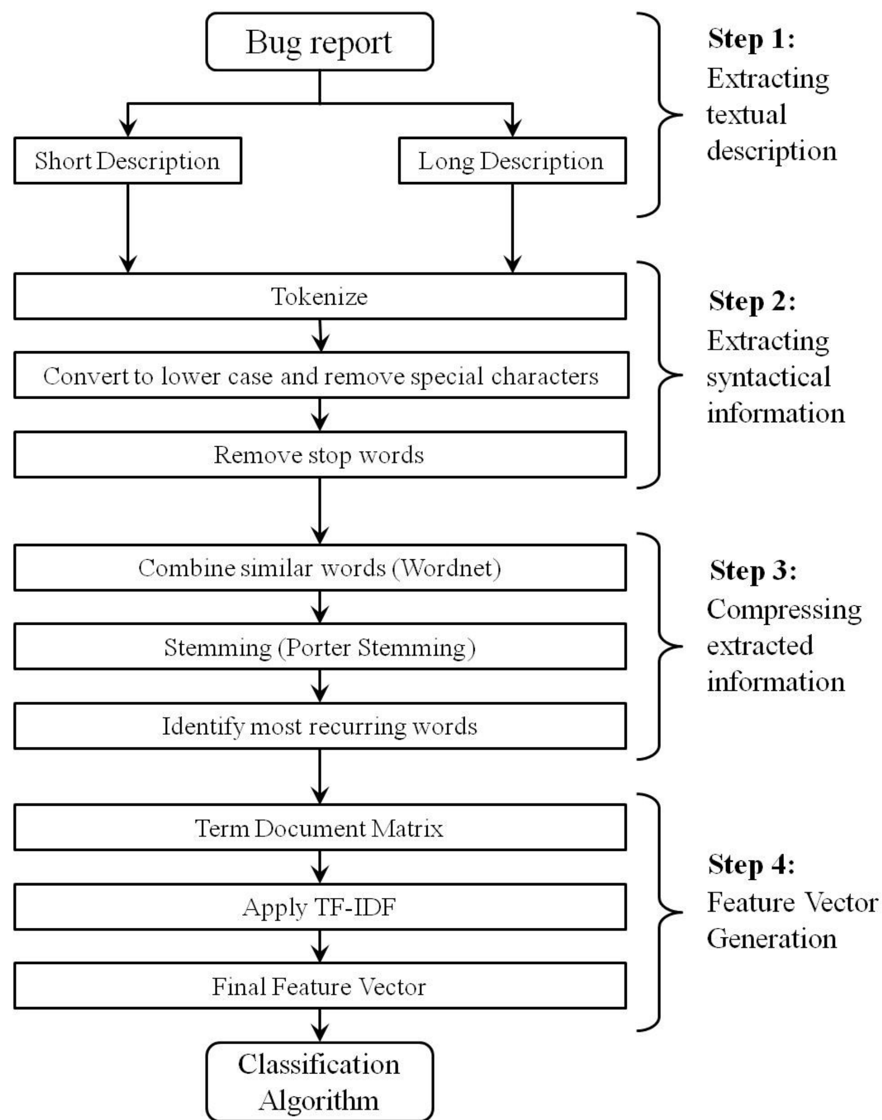

Figure 14. Text mining bug databases for HIBs architecture

3.3 TEXT MINING BUG DATABASES FOR IDENTIFYING VULNERABILITIES

This section presents a text mining framework for identifying vulnerabilities using bug reports in publically available bug databases. The four step process described in Section 3.2 is utilized to extract the most relevant information from textual description of bugs in bug reports and generate a feature vector that can be used as an input to classification algorithms. The overall architecture of the presented text mining framework is shown in Figure 14. Each step is discussed in detail below.

Step 1: In the first step of the framework, the textual description of the bug report is extracted. It has been shown that both short and long descriptions of the bug report contain important data that can yield information for various purposes [Lamkanfi 10], [Lamkanfi 11]. Furthermore, it has been shown that other field such as "severity" and "importance" may be incorrect for many bugs [Lamkanfi 10], [Lamkanfi 11], [Wijayasekara 12]. Therefore, only the short description and the long description of the bug report are used to extract information. Furthermore, since the same word may carry different information when appearing in the short description compared to the long description, two separate bags-of-word are kept for short and long descriptions.

Step 2: In the second step, as described in Section 3.2.2, tokenizing, removal of special characters, and removal of stop words is performed to both words from short and long descriptions of the bug reports. Furthermore, since bug reports sometimes contain code snippets, single characters may also be present in the tokenized set of words. These single character words are also removed in this step. Furthermore, as an additional step the words "*vulnerability*" and "*CVE*" are also removed. This is to alleviate any bias towards HIBs since these words may have been included in the bug report of HIBs.

Step 3: The third step of the framework is compressing the extracted information. In this step, first Wordnet [Fellbaum 98] is used as the English word database to generate synonyms and keywords are combined as explained in Section 3.2.3 to reduce the number of dimensions in the bag-of-words. Once the keywords are combined using synonyms each dimension of the bag-of-words representation is a set of keywords. Next, Porter Stemming [Porter 80] is used to stem the keywords sets into their basic form. After all keywords sets are stemmed, they are again combined and the number of dimensions is further reduced. Similar

to the previous step, Wordnet and Porter stemming is applied to the keywords from short description and long description separately. Finally, the most recurring keywords sets in the complete set of bug reports is found and only the top $T_{SD}\%$ of keywords sets are kept in the short description and the top $T_{LD}\%$ of keywords sets are kept in the long description.

Step 4: Finally, in the fourth step of the framework, a TDM is generated using the extracted set of keywords sets. As before, keywords sets from the short and long descriptions are treated separately. For each bug report, first each word in the short description is extracted. Each extracted word is then converted to lower case and all formatting is removed. Since the keyword sets in the bag-of-words are stemmed, the each extracted word is stemmed as well. These stemmed words are then compared to the words in the keyword sets in the bag-of-words for the short description. If the word is found, then the number of occurrences for that keyword set is incremented by one in the TDM. This process is repeated for all the words in the short description and long description separately to generate the complete TDM. Once the TDM is generated, it is further enhanced by applying the TF-IDF method described in Section 3.2.4.

Table 4. Dimensionality of the bag-of-words after each text mining step

Parameter	After tokenization	After removing stop words and other characters	After Wordnet	After Porter Stemming	After removing words that occur in less than 10% of bugs
Long Description	6161	6039	4536	4349	90
Short Description	9981	9843	8067	7825	158
Total	**16142**	**15882**	**12361**	**12174**	**248**

The presented text mining framework for information extraction and compression of textual information was applied to 1000 randomly selected Linux Kernel bugs from the Redhat Bugzilla bug database that were reported in the time period from January 2004 to April 2011. The number of keywords after each step of the text mining framework is shown in Table 4.

3.4 CONCLUSION

In this chapter the details of a comprehensive framework for extracting textual information was presented. The presented framework extracts relevant syntactical textual information from the short and long description of the bug reports. Furthermore, the extracted information is compressed with minimal loss of information and converted in to a feature vector that can be read by classification algorithms.

The presented method was applied to a small randomly selected set of bugs from the Redhat Bugzilla bug database, and the dimensionality reduction capability of each step in the presented text mining framework was demonstrated.

Chapter 4 Classification Of Hidden Impact Bugs

This chapter investigates different classification algorithms for classification of HIBs in bug databases. Three different classification algorithms will be investigated, namely: 1) Naïve Bayes (NB) classifiers, 2) Naïve Bayes Multinomial (NBM) classifiers, and 3) Decision Tree (DT) classifiers. The classification algorithms were applied to the HIB classification problem for the Linux Kernel. The chapter first provides background details of these algorithms. The details of the experiment as well as experimental results are provided next. The chapter is concluded by discussing the importance and relevance of the classification of HIBs.

4.1 Classification Algorithms

This section details the machine learning based classifiers that were used for the classification of HIBs. Three different classifiers were used for classification: 1) Naïve Bayes (NB) classifiers, 2) Naïve Bayes Multinomial (NBM) classifiers, and 3) Decision Tree (DT) classifiers.

4.1.1 Naïve Bayes (NB) And Naïve Bayes Multinomial (NBM) Classifiers

Naïve Bayes (NB) and Naïve Bayes Multinomial (NBM) classifiers are semi-interpretable probabilistic classifiers that have been shown to produce favorable results in text classification applications [Wen 07], [Lamkanfi 11], [Barber 12]. Both NB and NBM utilize the well known Bayes theorem for conditional probability [Barber 12]:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{4.1}$$

Where A and B are dependent events and $P(A \mid B)$ denotes the probability of event A given event B.

The NB and NBM classifiers utilize the Bayes theorem while assuming *naïve* independence of features in the input space [Barber 12]. Thus, for a given input vector d that belongs to the class C:

$$d = \{w_1, w_2, ..., w_m\} \tag{4.2}$$

which is analogous to equation (3.12) that represents the derived feature vector for a given document, using the Bayes theorem, the probability that the input vector belongs to class C can be written as:

$$P(C \mid d) = \frac{P(d \mid C)P(C)}{P(d)} \tag{4.3}$$

$$= P(C \mid w_1, w_2, ..., w_m) = \frac{P(w_1, w_2, ..., w_m \mid C)P(C)}{P(w_1, w_2, ..., w_m)} \tag{4.4}$$

However, the naïve conditional independence assumes that each feature w_i is conditionally independent from all other features w_j for $\forall j \in m$ and $i \neq j$. Thus, the numerator of equation (4.4) can be expressed as [Barber 12]:

$$P(w_1, w_2, ..., w_m \mid C)P(C) = P(C)P(w_1 \mid C)P(w_2 \mid C)....P(w_m \mid C) \tag{4.5}$$

$$P(w_1, w_2, ..., w_m \mid C)P(C) = P(C)\prod_{i=1}^{m} P(w_i \mid C) \tag{4.6}$$

Therefore, in the NB classifier, for a given set of documents, the conditional probabilities for each feature given each class $P(w_i \mid C)$ as well as the probability of each class $P(C)$ can be calculated. Thus, training data can be used to derive these probabilities in

a supervised manner. Once these probabilities have been derived, a given data pattern can be classified as:

$$class(d) = \arg\max_{c} P(C = c)\prod_{i=1}^{m} P(w_i \mid C = c) \tag{4.7}$$

However, since the NB classifier calculates the conditional probabilities $P(w_i \mid C)$ using the features of all documents, only binary cases of w_i can be used, i.e. the actual frequency of w_i cannot be used. Thus, NBM classifier uses a multinomial representation calculating the conditional probability $P(w_i \mid C)$ which enables the use of the frequencies of each feature for classification [Wen 07], [Barber 12].

$$P(w_i \mid C) = \frac{(\sum_i w_i)!}{\prod_i w_i} \prod_i p_i^{w_i} \tag{4.8}$$

where, p_i is the probability that event w_i occurs.

Furthermore, since the frequency of features are used in NBM, it has been shown that for many text classification applications, the classification accuracy of NBM is higher compared to NB [Wen 07], [Lamkanfi 11].

4.1.2 Decision Tree (DT) Classifiers

Decision Tree (DT) classifiers are multistage hierarchical decision support tools that are sequential in their approach and therefore highly interpretable [Safavian 91]. Thus DT can be visualized as a layered, directed graph of decisions. At each node of the graph the input space is divided into several crisp sub-spaces. This is done iteratively until a leaf node is reached, where the decision is the output class for the given input pattern. A typical binary decision tree, where at each node the input space is divided into 2 sub-spaces, is shown in Figure 15 [Safavian 91], [Hartmann 82].

Various methodologies and algorithms have been proposed for generating the optimal set of decisions, determining the number of decisions at each level, determining the depth of the tree, etc. [Safavian 91]. In interest of space, details of these methods will not be discussed in this thesis.

One of the most common methods of generating the set of decisions is using an Information Theoretic approach [Hartmann 82]. Information theory is briefly discussed in Chapter 5 of this thesis. In the Information Theoretic approach, the input space is sub-divided at each decision node using the dimension that yields the highest gain in information [Hartmann 82].

In order to reduce overtraining of DT as well as for computational complexity reasons, the length of the DT needs to be contained [Safavian 91]. While some methods limit the maximum depth, of the tree, several methods that prune a generated DT using various algorithms have been shown to be successful as well [Safavian 91].

DT classifiers rely on heuristic algorithms such as Genetic Algorithms (GA) for optimization of generated trees with respect to the number of nodes, the depth of the tree and classification accuracy (see Chapter 5 for more details on GA).

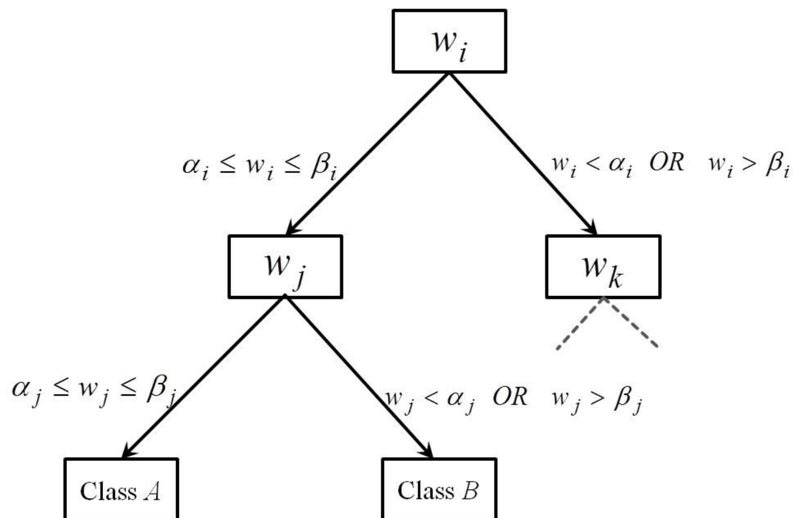

Figure 15. Typical binary Decision Tree (DT) classifier

The DT known as C4.5 which was developed by Quinlan is used in this thesis for classification [Quinlan 93]. The C4.5 DT is a simple yet proven methodology for deriving decision trees using information entropy. Furthermore, it has the capability of handling continuous values by means of using thresholds. For a given set of features, the C4.5 algorithm first calculates the normalized information gain for each feature. The algorithm then creates a splitting node which splits the dataset into two subsets based on the feature with highest information gain. This is recursively done until the subset contains only a single class or no information can be gained from the remaining subset of data, at which point a leaf node is inserted. The pseudocode for the C4.5 algorithms is shown in Figure 16 [Quinlan 93].

```
DTree(examples, features) returns a tree
{
   if (all examples are in one category)
   {
      return (leaf node with that category label);
   }
   else if (the set of features is empty)
   {
      return (leaf node with the most common category label);
   }
   else (pick a feature F and create a node R for it)
   {
      for (each possible value vi of F)
      {
         examplesi = subset of examples having value vi for F;
         Add an out-going edge E to node R labeled with the
         value vi;
         if ( examplesi is empty)
         {
            attach a leaf node to edge E labeled with the
            category that is the most common in examples;
         }
         else
         {
            call DTree(examplesi , features - {F}) and attach
            the resulting tree as the subtree under edge E;
         }
      }
      return (subtree rooted at R);
   }
}
```

Figure 16. Pseudocode for C4.5 algorithm

4.2 Classification Of HIBs In Bug Databases

The presented classification algorithms were tested on a subset of bug reports and HIBs for the Linux Kernel that were reported in the time period from January 2006 to April 2011. This section explains in detail the experimental setup that was used to test the presented HIB classification methodology as well as the test metrics used to evaluate the presented classifiers.

4.2.1 Classification Subset

In order to evaluate the vulnerability identification methodology proposed in Section 2.5, a set of Redhat Bugzilla bugs for the Linux Kernel, containing two classes: regular bugs and HIBs were constructed.

The MITRE CVE, vulnerability reports contain the bugs associated with each vulnerability. This information was used to extract bug reports from the Redhat Bugzilla bug database that were associated with the identified vulnerabilities. As shown in Section 2.4.2, out of the 403 vulnerabilities that were examined, 129 were identified to be HIBs with at least 2 weeks of impact delay. However, out of the 129 HIBs, only 73 had accessible bug reports in the Redhat Bugzilla bug database attached with them. The remainder had either no bug reports associated with them, or bug reports were inaccessible, or bug reports were in a different bug database. Therefore, for the final classification and testing the set of 73 identified HIBs were used. These bugs constitute the HIB class.

The regular bug class contained 6000 randomly selected bugs reported from January 2006 to April 2011 that were not identified as vulnerabilities. Since the number of bugs reported per year is different for each year (see Table 1), in order to avoid misrepresenting any year, the random set was constructed to reflect the proportion of bugs reported for each year. However, it is important to note that the regular bug class may contain bugs that are misclassified and are still yet to be identified as vulnerabilities, and the classifiers may be negatively affected by training on these examples [Jason 13].

Table 5 shows the number of HIBs that were identified for each year and the number of regular bugs selected from each year for the classification process.

4.2.2 CONSTRUCTION OF THE TDM

The Term-Document Matrix (TDM) was constructed using the short and long descriptions of the bug reports. As mentioned in Section 3.3, the short and long descriptions of the bugs were treated separately, meaning the text mining process was applied to words extracted from the short description and the long description separately.

The percentages T for selecting the keywords appearing most frequently in bug reports were set at T_{SD} = 1% and T_{LD} = 3%. These numbers were selected somewhat arbitrarily, so that a reasonable number of dimensions will be selected without overwhelming the classifiers. The same thresholds were used to test each classifier. Although this type of arbitrary dimensionality selection is sub-optimal for classification, it was deemed sufficient for demonstrating the vulnerability identification methodology described.

The text mining process which was elaborated in Chapter 3 was applied to the extracted long and short descriptions of bugs and the dimensionality of the TDM after each step is shown in Table 6.

Table 5. Number of selected regular bugs and HIBs for classification

Year	Number of regular bugs	Number of HIBs
2006	642	3
2007	725	12
2008	876	21
2009	1,135	10
2010	1,819	25
2011 (to April)	803	2
Total	**6,000**	**73**

Table 6. Dimensionality of the bag-of-words after each text mining step

Parameter	After removing stop words and other characters	After Wordnet	After Porter Stemming	Selecting Most frequent
Long Description	7,279	4,451	4,005	66
Short Description	27,685	20,800	19,113	136
Total	**34,964**	**25,251**	**23,118**	**202**

4.2.3 TEST METRICS

The classification was performed using *k*-fold cross validation [Wijayasekara 11]. The *k*-fold cross validation method randomly separates the dataset into *k* equal sized folds. The classifiers are then trained using *k* - 1 folds and tested on the remaining fold. This process is repeated *k* times, each time selecting a different fold for testing and using the remaining folds for training. The final classification result is an aggregation of the testing folds. This enables the classifier to be tested on all the data points and is therefore devoid of over fitting and biased classification [Wijayasekara 11]. 10-fold cross validation was used for testing the chosen classifiers.

Classification results are shown in terms of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). These parameters can be interpreted using the confusion matrix shown in Table 7. The following rates are calculated in order to present the results more accurately:

$$TP\ Rate = \frac{TP}{(TP + FN)} \tag{4.9}$$

$$TN\ Rate = \frac{TN}{(TN + FP)} \tag{4.10}$$

$$FP\ Rate = \frac{FP}{(TN + FP)} \tag{4.11}$$

$$Fn\ Rate = \frac{FN}{(TP + FN)} \tag{4.12}$$

Table 7. Confusion matrix for classification of HIBs

		Classified as	
		HIB	Regular
Actual Class	HIB	True Positives (TP)	False Negatives (FN)
	Regular	False Positives (FP)	True Negatives (TN)

4.2.4 BAYESIAN DETECTION RATE

Another important classification metric that was used to evaluate the classifiers was the Bayesian detection rate [Axelsson 00]. The Bayesian detection rate is the probability that an instance classified as true, is actually true [Axelsson 00].

The Bayesian detection rate is extremely important in this case due to the disproportionate sizes of the two classes. In [Axelsson 00] Axelsson performed a base-rate fallacy test for intrusion detection systems and illustrated the problems in classifying intrusions. Axelsson pointed out the small ratio between the number of intrusions and normal traffic affect the outcome in such a way that the user will be overwhelmed by the number of false positives. Since the ratio between HIBs and normal bugs in bug databases is very low:

$$\frac{129}{167{,}390} = 7.71 \times 10^{-4} \tag{4.13}$$

where, 167,390 is the number of normal bugs reported for the time period between January of 2004 and April of 2011, a similar base-rate fallacy problem may occur. Thus a base-rate fallacy analysis was performed for the HIB classification problem. For explaining the base-rate fallacy, the following nomenclature will be used:

$V = \textit{hidden impact vulnerability}$
$D = \textit{detection} \ (\textit{i.e. the classifier detects a bug as a HIB})$
$\neg X = \textit{not } X$
$P(X) = \textit{probability of } X$
$P(X \mid Y) = \textit{probability of } X \textit{ given } Y$

By using the above naming convention, true positive rate can be denoted as $P(D \mid V)$ and the false positive rate can be denoted as $P(D \mid \neg V)$.

For classification of HIBs the Bayesian detection rate is the probability that a bug is a HIB given that the classifier detects the bug as a HIB, i.e. $P(V \mid D)$. In order to increase the Bayesian detection rate, the number of false positives must be reduced. By means of Bayes' theorem the Bayesian detection rate can be expressed as:

$$P(V \mid D) = \frac{P(V) \bullet P(D \mid V)}{P(V) \bullet P(D \mid V) + P(\neg V) \bullet P(D \mid \neg V)} \tag{4.14}$$

The following probabilities are known:

$$P(V) = \frac{129}{167390} = 7.71 \times 10^{-4} \tag{4.15}$$

$$P(\neg V) = 1 - P(V) = 1 - 7.71 \times 10^{-4} = 0.99923 \tag{4.16}$$

By using equations (4.15) and (4.16), equation (4.14) can be rewritten as:

$$P(V \mid D) = \frac{7.71 \times 10^{-4} \bullet P(D \mid V)}{7.71 \times 10^{-4} \bullet P(D \mid V) + 0.99923 \bullet P(D \mid \neg V)} \tag{4.17}$$

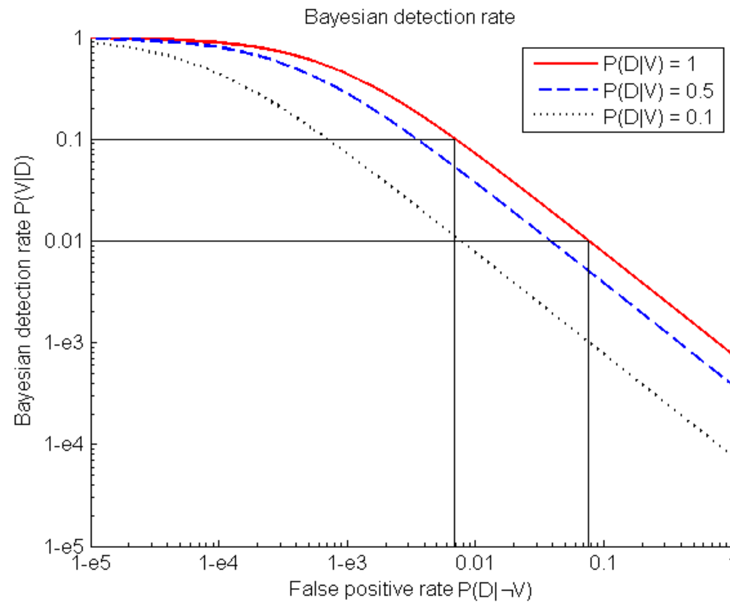

Figure 17. Bayesian detection rate for the HIB classification problem

The Bayesian detection rate expressed in equation (4.14) is dominated by the factor 0.99954, i.e. the high probability that a bug is not a HIB. Thus in order to achieve a Bayesian detection rate that is sufficient, the false positive rate must be very low. Figure 17 plots the false positive rate against the Bayesian detection rate for different values of true positive rates ($P(D|V)$). Figure 17 shows that as the false positive rate increases, the Bayesian detection rate decreases (Note that the axis are in log scale).

The Bayesian detection rate is vital when dealing with human users such as a software development team: if the Bayesian detection rate is too low, the users will be overwhelmed by the number of false positives and thus reducing the effectiveness of the classifier [Wijayasekara 12]. By using the upper bound in Figure 17, it is possible to gain an understanding of the maximum false positive rate which is acceptable from the classifier. For example, if a Bayesian detection rate of 0.01 can be tolerated by the development team, which means that only one out of 100 bugs classified as potential vulnerabilities, is an actual vulnerability, according to Figure 17, a maximum false positive rate of 0.076 is acceptable. This means that on average for any given day in 2011, where 146 bugs were reported per day (see Table 1), around 11 (0.076×146) bugs will be falsely identified as a vulnerability by the classifier. Similarly, if one out of 10 detections needs to be an actual vulnerability, which means a Bayesian detection rate of 0.1, to achieve this, the maximum acceptable false positive rate is 0.0069. This translates to falsely identifying around one bug per day (0.0069×146) for any given day in 2011. Thus, the lower boundary of false positive rate that the proposed classifier must obtain can be determined using Figure 17.

Thus, a higher Bayesian detection rate means that a lower percentage of regular bugs were improperly classified as HIBs and therefore a software development team will have to sort through fewer regular bugs to find those which are actual vulnerabilities. Therefore, a higher Bayesian detection rate is preferred. Given the classification results, the Bayesian detection rate can be calculated using the following equation:

$$Bayesian\ \ Detection\ \ Rate = \frac{TP}{(TP + FP)} \tag{4.18}$$

Furthermore, with increasing numbers of bugs being reported each day (see Table 1), a low Bayesian detection rate will lead to an overwhelming number of bugs being classified by the classifier as potential vulnerabilities each day. Using the average number of bugs reported per day, the number of bugs that will be classified as potential vulnerabilities by the classifier can be calculated as:

$$\frac{TP + FP}{(TP + FP + TN + FN)} \times Bugs\ Reported\ \ per\ Day \tag{4.19}$$

Thus, the number of bugs that will be classified as potential vulnerabilities can also be used at a metric for evaluating classifier performance.

Table 8. Overall classification results

Classifier	True positives		True negatives		False positives	
	Rate	Number	Rate	Number	Rate	Number
Naïve Bayes (NB)	0.92	67	0.45	2,741	0.55	3259
Naïve Bayes Multinomial (NBM)	0.81	59	0.90	5,377	0.10	623
Decision Tree (DT)	0.29	21	0.99	5,969	0.01	31

Table 9. Number bugs classified as of potential vulnerabilities on a given day in 2011

Classifier	Number of bugs classified as potential vulnerabilities per day	Bayesian detection rate
Naïve Bayes (NB)	80.4	0.02
Naïve Bayes Multinomial (NBM)	16.5	0.09
Decision Tree (DT)	1.3	0.40

4.2.5 Classification Results

The overall classification results are shown in Table 8. NB classifier showed the highest true positive rate (92%), however, the true negative rate was low. Similarly the DT had a very low true positive rate (29%) but the highest true negative rate (99%). NBM showed a higher true positive rate as well as higher true negative rate. Although these results may seem relatively low, even the lowest true positive rate (29%) is more than 20 times better than a random guess (1.2%).

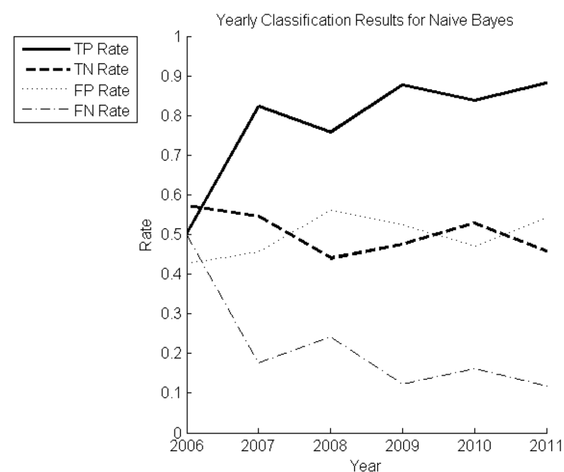

Figure 18. Yearly classification results for NB classifier

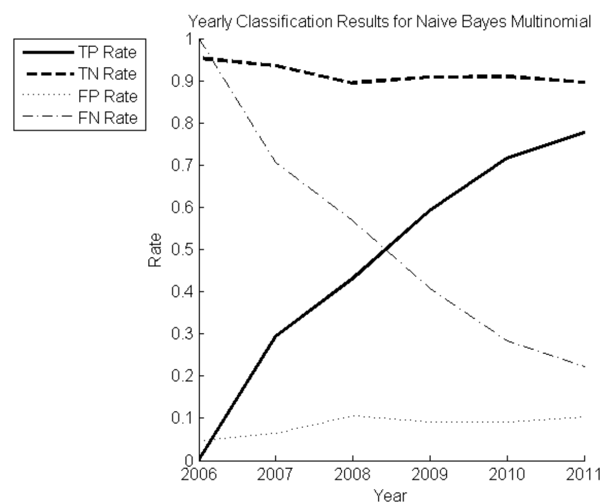

Figure 19. Yearly classification results for NBM classifier

Table 9 shows the number of bugs that will be classified as potential vulnerabilities on an average day in 2011 for each classifier along with the Bayesian detection rate. These results show that although the DT had a low true positive rate, due to the higher Bayesian detection rate nearly half of the bugs that are classified as potential vulnerabilities are actual vulnerabilities. This also leads to less than 2 bugs being classified as potential vulnerabilities each day in 2011 where over 145 bugs were reported to the bug database each day (see Table 1).

Similarly, NBM classifier reported a Bayesian detection rate of 0.09, which means that just under one out of 10 bugs that are classified as a potential vulnerability is an actual vulnerability. Using the NBM classifier, 16 bugs will be classified as potential vulnerabilities each day in 2011.

Due to the very high false positive rate of the NB classifier, the Bayesian detection rate was extremely low (0.02), which translates to only 1 out of 50 bugs that are classified as potential vulnerabilities being actual vulnerabilities. This also means that 80 bugs would be classified as potential vulnerabilities each day, in 2011, using the NB classifier.

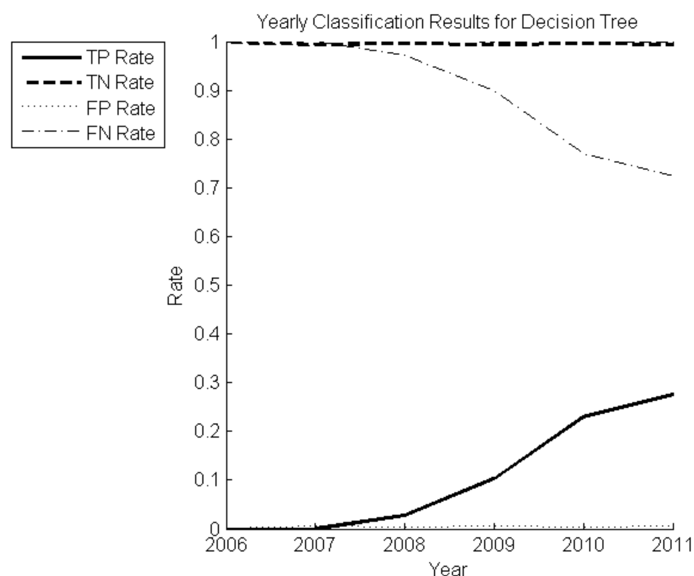

Figure 20. Yearly classification results for DT classifier

According to Table 5, the number of reported bugs and the number of identified HIBs have been increasing each year. Thus, in order to evaluate the usability of the classifiers in this real world scenario, and to evaluate the online learning capability of the system presented in Section 2.5.2, the performance of each classifier was measured across time, based on the data available at a given moment in time. For this analysis, cumulative bugs reported and HIBs found at the start of each year was selected.

Figure 18, Figure 19, and Figure 20 plot the yearly classification results for NB, NBM and DT classifiers respectively. As expected, the true positive rate increases with time. This is because the size of the HIB set is increasing and the classifiers are able to learn from these. Therefore, as more HIBs are correctly classified as vulnerabilities, the classifiers benefit from these newly discovered HIBs.

4.3 Conclusion

This chapter first detailed the classification algorithms used to classify HIBs using the framework presented in Chapter 2 and Chapter 3.

The chapter then detailed the experimental setup used for evaluating the classifiers and the presented HIB classification and text mining frameworks. General classification metrics as well as classification metrics specific to the HIB classification problem was discussed next. A Bayesian detection rate analysis was performed to identify the upper and lower bounds of classification accuracy required by the classifiers.

Finally, the experimental results for each of the classifiers were elaborated. The tested classifiers were able to correctly classify 29% to 92% of the HIBs in the Linux Kernel. The lowest achieved classification rate was over 20 times better than a random guess. Further analysis on the Bayesian detection rate of the classifiers showed that the number of bugs that will be classified as potential vulnerabilities per day, given the results of each classifier. The results ranged from classifying 16 bugs per day as potential vulnerabilities, where 1 in 10 were actual vulnerabilities, to classifying 80 bugs per day as potential vulnerabilities, where only 1 in 50 were actual vulnerabilities. This information can be used by developers to select the optimal classifier, given the number of bugs that can be handled by a software maintenance team.

It has to be noted that the classification process was performed extremely conservatively, using keywords from only the set of regular bugs. Furthermore, it is important to identify that the set of regular bugs may contain HIBs that have not yet been identified which may further reduce the classification accuracy. Simultaneously, this means that the false positives of the classifiers may contain misclassified bugs that may turn out to be vulnerabilities. In order to verify this, the false positives should be further analyzed.

Chapter 5 INFORMATION GAIN BASED DIMENSIONALITY SELECTION

Classification is highly dependent on the dimensionality of the problem domain. Thus, selecting the optimal subset of dimensions can lead to higher classification accuracy as well as improved computational efficiency [Raymer 00], [Otero 03]. Text mining applications, in particular, where the extracted dimensionality is high, are extremely susceptible problems associated with sub-optimal dimensionality selection [Basiri 09]. Therefore, this Chapter presents a novel dimensionality selection method that is based n information theory.

The Chapter first discusses the broader dimensionality selection problem and shows the necessity of dimensionality selection in the HIB classification problem. Then, background information about Genetic Algorithms (GA) and Information Gain (IG) are presented. The novel, information gain based dimensionality method is detailed next. The presented method was applied to the problem of text mining bug database for identifying HIBs, and the experimental setup and results are detailed next. Finally, the Chapter is concluded by discussing the importance of dimensionality selection and other possible improvements to the presented methods.

5.1 DIMENSIONALITY SELECTION PROBLEM IN TEXT MINING

Dimensionality selection is used as an important step in knowledge extraction and data mining applications for better understanding of data [Raymer 00], [Battiti 94]. Proper usage of dimensionality selection methodologies can result in lower computation time and achieve higher classification accuracy in classification applications [Raymer 00], [Otero 03]. These methodologies are especially useful highly multi dimensional datasets where the high dimensionality increases computation time significantly.

Typical text mining applications investigate large number of documents and extract syntactical information by means of unique words occurring in the documents [Wijayasekara 12]. As shown in Table 4, this type of information extraction results in highly multi

dimensional datasets with a sparse matrix. Thus, dimensionality selection methodologies are employed in text mining applications to identify the optimal set of dimensions that yield the best classification results possible [Basiri 09].

Dimensionality selection (or feature selection) is a form of transformation of representation [Liu 98], where a set of dimensions M, is derived from the original set of dimensions M_0 that maximizes some criterion and is at least as good as M_0 in that criterion [Jain 97]. In classification applications, the maximization criterion is the classification accuracy [Jain 97].

Dimensionality selection has been successfully performed using Genetic Algorithms (GA) for text mining [Espejo 10], [Sebastiani 02], [Yang 97] and other applications [Otero 03], [Espejo 10], [Yang 97]. Information Gain (IG) [Shannon 48] has also been used successfully in-conjunction with GA for dimensionality selection in classification and other data mining problems [Sebastiani 02], [Yang 97]. However, these studies use IG as either a data pre-processing step [Uguz 11] or as the fitness function of the Genetic Algorithm [Otero 03], [Basiri 09], [Neshatian 08], [Muharram 05].

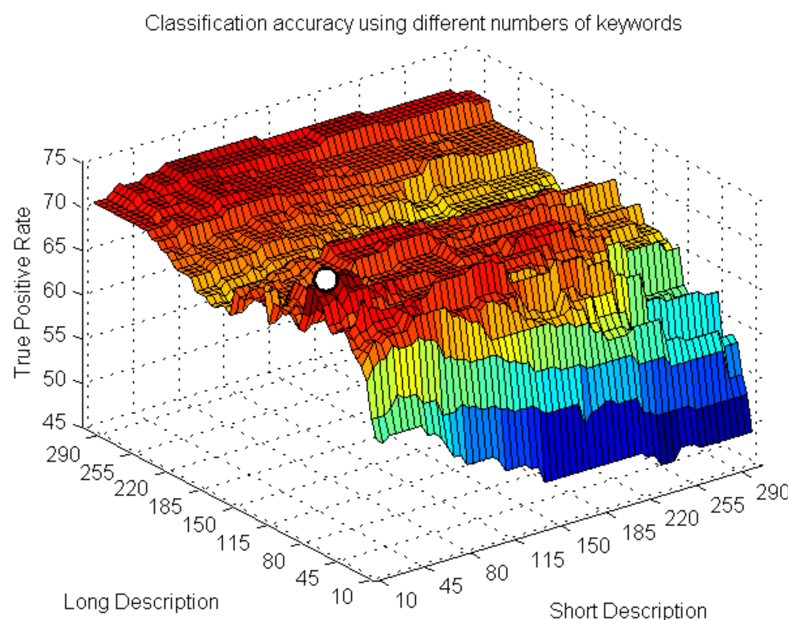

Figure 21. Averaged true positive rate for different numbers of keywords

5.1.1 DIMENSIONALITY DEPENDENT CLASSIFICATION ACCURACY

As mentioned, classification accuracy highly depends on the selected set of keywords. In order to illustrate this, several classifiers were used to classify 1000 randomly selected bugs from the Redhat Bugzilla bug database and the 73 HIBs identified in Section 4.2.1, using different number of keywords.

The NBM and C4.5 classifiers described in Sections 4.1.1 and 4.1.2, respectively were used to classify the above mentioned dataset. For training and classification, different numbers of keywords from the short description and the long description were selected.

The averaged classification results of the NBM and C4.5 Classification algorithms were measured. Figure 21, Figure 22 and Figure 23, respectively plot the averaged true positive rate, true negative rate and the average of true positive and true negative rates as the classification accuracy. The best result for each graph is emphasized using a white dot.

It can be observed from the figures that the optimal value for each classification metric occurs at a different number of keywords used. Furthermore, contrary to intuition, adding more dimensions sometimes reduces the classification accuracy. In Figure 21, it can be seen that adding the 200th dimension in the long description significantly reduced true

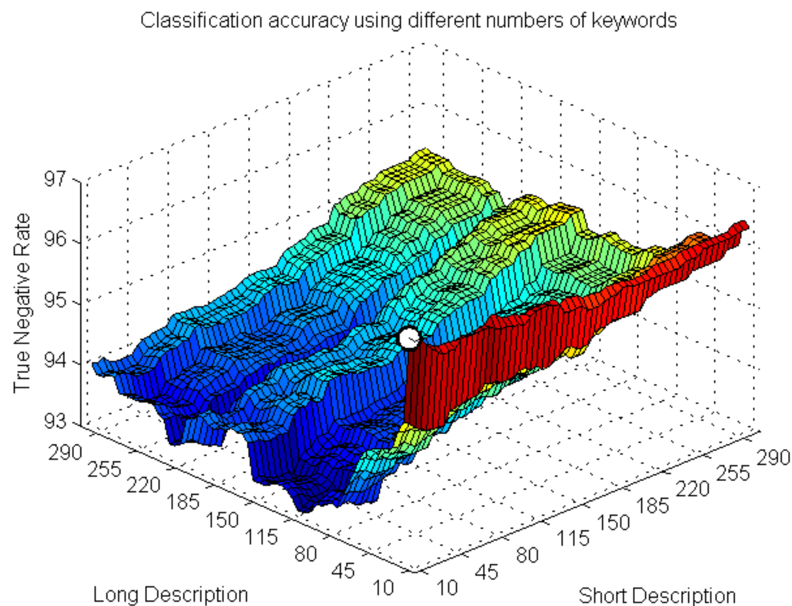

Figure 22. Averaged true negative rate for different numbers of keywords

positive rate. It has to be noted that this dimension was added while all the previous dimensions that caused the higher true positive rate remained in the TDM. This is largely because adding dimensions that does not contribute to classification or dimensions that affect the classification negatively, lessen the ability of the classifier to focus on the attributes that contribute to the classification more.

Furthermore, different combinations of keywords may yield different results. This aspect is cannot be observed in the figures as different combinations of dimensions were not tested. Thus, selecting the optimal dimensionality is a highly non-linear multiple criteria optimization problem.

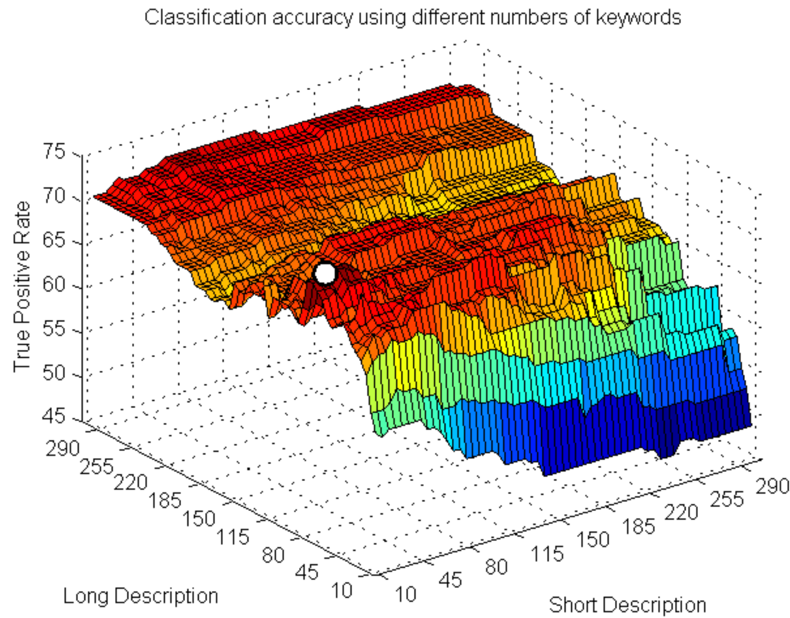

Figure 23. Averaged classification rate for different numbers of keywords

5.2 BACKGROUND

This section first details Genetic Algorithms (GA) and then presents background information on information theory and Information Gain (IG).

5.2.1 GENETIC ALGORITHMS (GA)

Genetic Algorithms (GA) are a subset of the broader field of Evolutionary Algorithms (EA). The major unifier of EA is the application of simulated biological evolution. Simulated evolution is inspired by and analogous to the well known Darwin's theory of evolution, and has been translated into an effective tool for global optimization [Goldberg 89], [Simon 13], [Linda 14].

The common underlying idea is that the algorithm maintains a set of unique candidate solutions to the problem which are comparative to a set of individuals in a population. The ability of each solution or individual to solve the problem can be evaluated based on an objective fitness function, and is known as the fitness of an individual. This fitness is subsequently used drive the evolution of the population based on the theories of natural selection [Simon 13], [Linda 14]. Thus, at each iteration, the fitness of every individual is calculated and based on the fitness, certain individuals are removed from the population and new individuals are introduced.

In Genetic Algorithms (GA), biological genetics analogies are used to further advance the evolutionary process of EA [Simon 13]. This is done by encoding the candidate solutions are represented as chromosomes using a set of real numbers:

$$v = \{r_1, r_2, ..., r_k\} \tag{5.1}$$

where, v is an individual and r_i is a real value. The candidate solution for the problem is a combination of the real values and the fitness of the individual is calculated using:

$$\widetilde{f}_v = f(r_1, r_2, ..., r_k) \tag{5.2}$$

where, $\widetilde{f}_v$ is the fitness of individual v and f is the fitness function

Thus, at each iteration, a certain set of individuals are selected based on their fitness and using those individuals as parents, offspring are generated. The offspring are generated from the selected parent individuals by applying recombination and mutation operators to the chromosomes of the parent individuals [Simon 13], [Linda 14].

The recombination operator splits the chromosomes of two parents at a certain point and combines each portion to generate two new individuals. The point at which the recombination happens can be selected using various domain specific heuristic methods [Simon 13].

```
GA(NumIndividuals, NumIterations) returns a solution
{
    Population P = NumIndividuals random solutions;
    Evaluate (P);
    iteration = 0;
    while (convergence is not met AND iteration < NumIterations)
    {
        parents = Tournament(P, TounamentSize);
        offspring = Recombination of parents chromosomes;
        offspring = Mutation of offspring;
        Evaluate(offspring);
        Insert offspring  to P;
        Evaluate (P);
        iterations = iterations + 1;
        Check convergence;
    }
    return individual with best fitness;
}

Tournament(Population, TounamentSize) returns individuals
{
    Select subset from Population with the size of TounamentSize;
    return two individuals with the best fitness;
}
```

Figure 24. Pseudocode for Genetic Algorithm (GA)

The mutation operator mutates a selected individual by randomly selecting one or more values from $\{r_1, r_2, \ldots, r_k\}$ and changing these values by a small random value. The mutation point selection and the amount to which the mutation is performed can also be done using domain specific knowledge [Simon 13].

This cycle is repeated for a specified number of iterations or until another convergence criterion is met, such as the desired level of the best fitness value or the standard deviation of the fitness value within the population. The general pseudocode of GA is summarized in Figure 24 [Simon 13], [Linda 14].

5.2.2 Information Gain (IG)

Information theory was founded by Claude Shannon in 1948 [Shannon 48] and has since been used as the primary methodology of information quantization of datasets. One of the basic foundations of information theory is the concept of information entropy. Using the information entropy of a dataset and the information entropy of the same dataset given that a dimension is known, can be used to derive how much information is gained by knowing that dimension. This is known as the Information Gain (IG) of that dimension.

The information entropy of a dataset defines the distribution of the dataset in classes. Higher information entropy describes a uniform class distribution meaning more information is required to identify each class separately. Similarly lower information entropy describes a variable class distribution and less information is required to identify each class [Wenke 01]. The information entropy of a dataset D can be calculated using:

$$Entropy(D) = -\sum_{j=1}^{c} p_j \times \log_2 p_j \tag{5.3}$$

where, c is the number of distinct classes in D and p_j is the proportion of instances in D that belong to class j [Shannon 48]. Similarly the information entropy of a subset of the dataset can be calculated as:

$$Entropy(\widetilde{D}) = -\sum_{j=1}^{c} q_j \times \log_2 q_j \tag{5.4}$$

where, $\widetilde{D}$ is a subset of the dataset D and c is the total number of distinct classes in D and q_j is the proportion of cases in $\widetilde{D}$ that belong to class j.

Furthermore, the information entropy of the dataset D can be calculated given the dimension w is known:

$$Entropy(D \mid w) = -\sum_{j=1}^{K} \frac{n_k}{N} \times Entropy(\widetilde{D}_k) \tag{5.5}$$

where, N is the total number of data points in the dataset D, and K is the number of distinct partitions caused by dimension w. n_k is the number of cases in D that belong to the partition k and $\widetilde{D}_k$ is the partition of data caused by k. The entropy of $\widetilde{D}_k$ is calculated using equation (5.4) [Shannon 48].

The entropy of the dataset given that a dimension is known calculated using equation (5.5), shows the additional amount of information required to identify each class separately. Thus $Entropy(D)$ and $Entropy(D \mid w)$ can be used to calculate the additional information gained by knowing dimension w:

$$IG(w) = Entropy(D) - Entropy(D \mid w) \tag{5.6}$$

where, $IG(w)$ is the information that can be gained if dimension w is known [Wenke 01]. Thus, Information Gain (IG) of any one dimension of the dataset is independent from any other dimension in the dataset.

5.3 IG BASED DIMENSIONALITY SELECTION FOR TEXT MINING APPLICATIONS

Selecting the optimal set of dimensions, as pointed out in Section 5.1, is essential for achieving improved classification rates and computational efficiency. Therefore, this section presents a novel methodology dimensionality selection that utilizes the IG of each dimension in the dataset to calculate dynamic mutation probabilities for chromosomes in a GA [Wijayasekara 13]. Thus the probability of selecting or deselecting a given dimension at each mutation step is dependent on the IG of that dimension. This dynamic selective mutation favors dimensions with higher IG and enables the GA to converge to a more optimal solution faster. Furthermore, since IG of each dimension is independent from any other dimension in the dataset, IG can be calculated prior to the execution of the genetic algorithm. This leads to the computation time of presented IG based method to be the same as conventional GA based methods.

Typical genetic algorithm based dimensionality selection encodes the dimensions of the dataset as bits in a chromosome:

$$v_x = \{r_1, r_2, ..., r_k\} \tag{5.7}$$

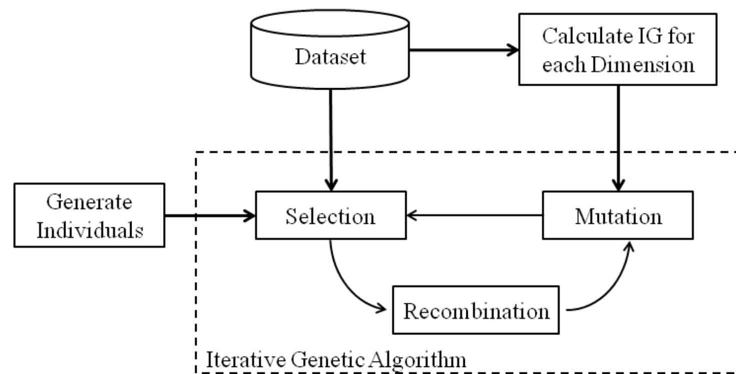

Figure 25. Block diagram of the presented IG based dimensionality selection method

where, v_x is the chromosome of individual x, and D is the number of dimensions in the dataset. r_i is a bit that represents whether dimension i is selected or not. At each iteration of the GA, the chromosome of an individual may change during recombination or mutation phases. This enables the population to evolve, and eventually reach an optimum, where the most optimum set of dimensions are selected by the individual with highest fitness. Therefore, for classification problems the fitness of an individual is calculated by the classification accuracy achieved by classifying the dataset using only the dimensions selected by the chromosome of that individual.

The presented IG based dimensionality selection methodology utilizes IG of each dimension to dynamically vary the mutation probability of chromosomes. The mutation probabilities are dynamically varied such that it favors the dimension with a higher IG. Since a dimension with higher IG means that more information about the class separation is gained by using the said dimension, such a selective mutation enables the genetic algorithm to reach the optimal value faster. A simple block diagram of the presented methodology is shown in Figure 25 [Wijayasekara 13].

As shown in Section 5.2.2, the IG of a dimension is independent from any other dimension in the dataset. Thus IG can be calculated for each dimension prior to the execution of the GA. Once the IG is calculated, it is normalized between 0 and 1 using:

$$IG(w_i) = \frac{IG(w_i) - IG(\min)}{IG(\max) - IG(\min)} \tag{5.8}$$

where, $IG(\min)$ and $IG(\max)$ are minimum and maximum information gain for all the dimensions in the dataset D, respectively.

This calculated IG is then used to dynamically vary the mutation probability of each dimension using:

$$p(v_x, i) = \left[IG(w_i) \times (p_{\max} - p_{\min})\right] + p_{\min} \tag{5.9}$$

Or,

$$p(v_x,i)=\left[(1-IG(w_i))\times(p_{\max}-p_{\min})\right]+p_{\min} \quad (5.10)$$

where, $p(v_x,i)$ is the probability that the ith bit of individual v_x is mutated, and $IG(w_i)$ is the normalized IG of dimension i calculated using equation (5.8). $p_{\min}$ and $p_{\max}$ are preset probabilities that define the maximum mutation probability and minimum mutation probability respectively, and are set such that $p_{\max} > p_{\min} \geq 0$. If bit i of individual v_x is 0, meaning the dimension is currently deselected, equation (5.9) is used to calculate the mutation probability and equation (5.10) is used otherwise.

Therefore, a dimension with higher IG has a higher probability of being selected at each mutation step. Similarly, such a dimension has a lower probability of being deselected during mutation.

5.4 Experimental Results

The presented IG based dimensionality selection method was applied to the HIB classification problem in the Linux Kernel. A similar dataset of bug reports containing known HIBs and bugs that were not identified as vulnerabilities, used in Section 4.2.1 was used in this Section as well. The dataset consisted on 73 HIBs and 6000 randomly selected normal bugs (See Table 5). The 6000 normal bugs were selected according to the respective proportions of bugs reported in each year of the time period from January 2006 to April 2011.

The text mining process described in Section 3.3 was used to extract the set of key-words that contain the most information and most relevant to the selected dataset. By selecting the 500 most frequent key-words from the short description and the long description of the bug report, a feature vector of the length 1000 was extracted (See Sections 2.3.1 and 3.2.4 for more details).

The presented methodology was compared to conventional GA based dimensionality selection method by using a generational GA. This type of GA uses recombination as part of the evolutionary process, along with mutation. In generational GA, a population consisting of

a certain number of individuals is kept, and at each iteration, parent individuals are selected by means of a tournament within a subset of the individuals in the population.

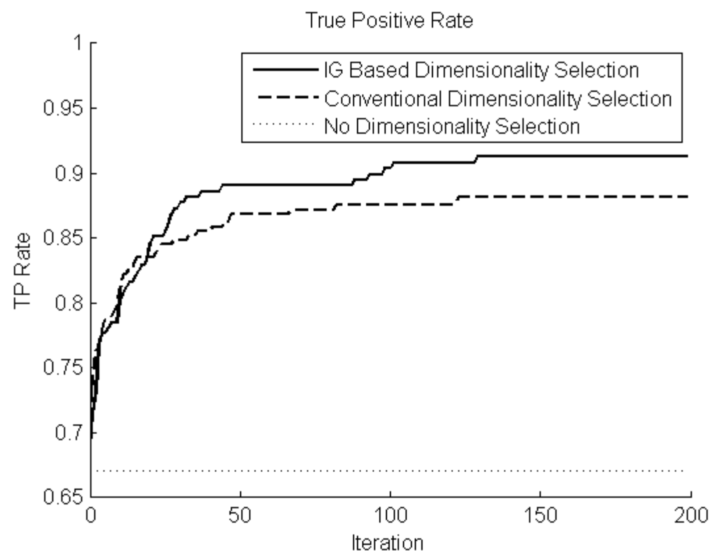

Figure 26. Averaged true positive rate for each iteration for each method

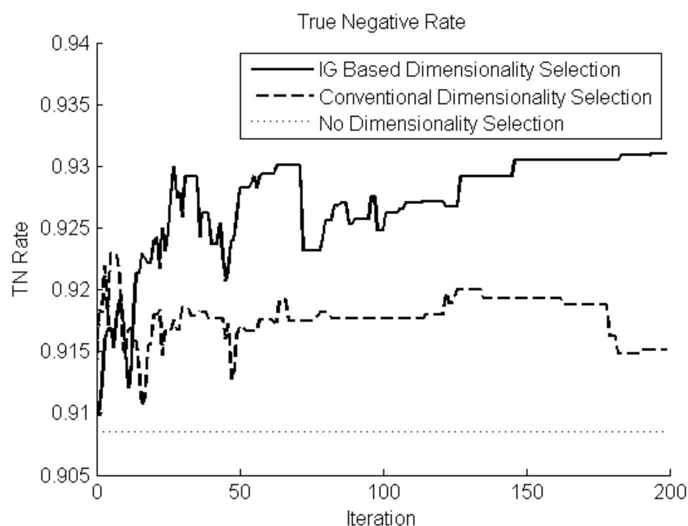

Figure 27. Averaged true negative rate for each iteration for each method

Both the GAs were tested using 50 individuals, with a tournament size of 10. The minimum and maximum mutation probabilities ($p_{\min}$, $p_{\max}$) for the presented IG based method were set at 5% and 10% respectively while the mutation probability of the conventional GA was set at 10%. The true positive rate and the true negative rate using Naïve Bayes Multinomial (NBM) classifier with 10-fold cross validation was used as the fitness function of both GA:

$$\tilde{f}_v = \frac{TP\ Rate_v + TN\ Rate_v}{2} \quad (5.11)$$

This fitness function was used to alleviate fitness bias towards either true positives or true negatives.

The GA were run for 200 iterations. Each method was executed 10 times with a different random starting population and the final results were averaged. The averaged true positive rates, true negative rates and Bayesian detection rates for each method, at each iteration, are shown in Figure 26, Figure 27, and Figure 28 respectively

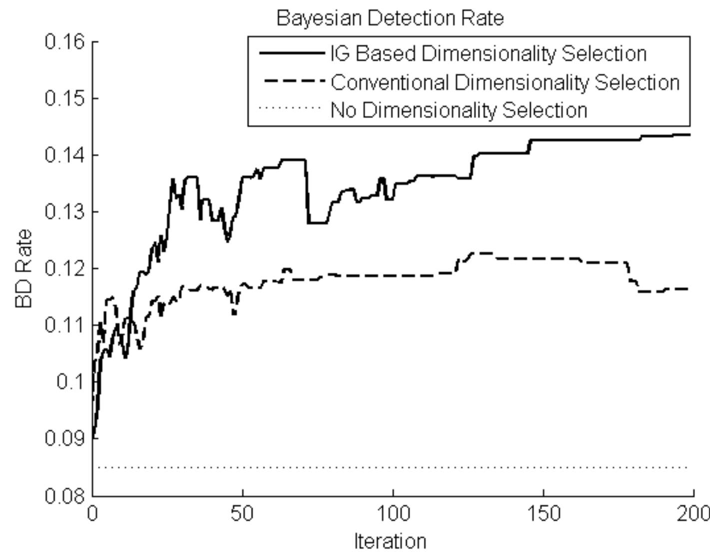

Figure 28. Averaged Bayesian detection rate for each iteration for each method

Table 10 shows the averaged final classification results with no dimensionality selection. Both dimensionality selection methods performed better than when the full 1000 dimensions are used. The presented IG based method shows more than 3% improvement over the conventional dimensionality selection method for true positive rate and the Bayesian detection rate and an improvement of 1.5% for true negative rate.

Table 10. Averaged classification results for each method.

Parameter	No dimensionality selection	Conventional GA based dimensionality selection	Presented IG based dimensionality selection
True Positive Rate	0.671	0.881	**0.912**
True Negative Rate	0.908	0.915	**0.931**
Bayesian Detection Rate	0.085	0.116	**0.143**

5.5 CONCLUSION

This Chapter presented a novel Information Gain (IG) based dimensionality selection methodology for text mining applications using Genetic Algorithms (GA). The presented methodology dynamically varies mutation probability of bits in the chromosome according to the IG of each dimension. This dynamic selective mutation enables selection of dimensions that contribute to classification more effectively.

The presented methodology was applied to the software vulnerability identification method discussed in this thesis. The presented methodology was applied to this text mining problem and compared with a conventional genetic algorithm with static mutation probabilities. The results show an increase of 3% for the true positives and the Bayesian detection rate and an increase of 1.5% for the true negatives in 200 iterations.

In addition to the presented application to the HIB classification problem, the novel IG based dimensionality selection method can be applied to other dimensionality selection problems as well.

Chapter 6 CONCLUSION AND FUTURE WORK

This Chapter provides the final conclusions of the presented work and proposes several directions for future work.

6.1 FINAL CONCLUSION

This thesis addressed the problem of identifying software vulnerabilities. In summary, a novel framework was developed, that extracts and utilizes textual information in publically available bug databases for identifying vulnerabilities. Several advancements of algorithms were developed for more efficient information extraction and classification of vulnerabilities.

First, Chapter 2 introduced the phenomenon known as Hidden Impact Bugs (HIBs) where a vulnerability is reported to a bug database as a bug before the full severity of that vulnerability is discovered. The Chapter then followed with an analysis of HIBs existing in 2 commonly used commercially available software packages, namely the Linux Kernel and MySQL Database Server. Chapter 2 also presented a novel framework for identifying software vulnerabilities, by leveraging information in bug databases. The presented framework extracts textual information of HIBs to classify whether a bug is a potential vulnerability or not, as it is being reported.

Chapter 3 presented a text mining architecture that is able to extract textual information from bug reports and convert the information in to a feature vector that can be utilized by classification algorithms. The presented architecture extracts syntactical information from bug reports and compresses the extracted information with minimal loss of knowledge.

Chapter 4 investigated several classification algorithms and applied these algorithms the presented textual information based vulnerability identification framework. The classification algorithms were used to classify HIBs in Linux Kernel. The classification results were then evaluated and the implications of these classification results in terms of real-world use of the framework by a software development team were discussed.

Finally, in Chapter 5 , a novel dimensionality selection methodology is presented for text document classification problems. The presented methodology utilizes relative information gain of keywords to drive the mutation probability of an evolutionary algorithm. The presented methodology was applied to the HIB classification problem for optimal dimension extraction.

6.2 Future Work

This section summarizes 7 primary directions for future work: 1) further validating the presented framework, 2) testing the presented framework on other software packages, 3) further increasing classification accuracy, 4) understanding what key-words contribute to the classification, 5) applying and validation of the developed dimensionality selection method on other problem domains, 6) developing classification framework that is less susceptible to dimensionality, training data set, and classifier, and 7) using the information gathered from identifying vulnerabilities to increase the security of future software releases.

In order to accurately validate the presented framework, the set of bugs that are classified by the framework as potential vulnerabilities must be examined to identify whether these bugs are actual vulnerabilities. This can be done by either domain experts who will closely examine the for possible security impact of these bugs, or by observing whether these bugs will later be identified as vulnerabilities within a given time period. However, since some bugs may be exploitable even though experts are unable to find vulnerabilities and as shown in Chapter 2 some HIBs may take over an year to be correctly identified as vulnerabilities correctly validating the presented framework will be difficult.

The presented framework should be tested on other commonly used software packages. Although, as mentioned in Chapter 2, Linux Kernel and MySQL Database Server are commonly used software packages that represent common commercial software, the presented framework should be tested on a diverse set of software packages to fully identify the capabilities and shortcomings of the presented framework. However, as mentioned in Chapter 2, public information about software vulnerabilities is scarce and some commercial vendors are reluctant to divulge such information.

The classification accuracy of the presented framework can be improved in two different ways. First, the text mining methodologies can be improved to include inter-relationships between words and differentiate certain multiword terms. However, this may lead to a significant increase of the possible permutations for generating the feature vector. Second, additional sources of information can be used to complement the classification obtained by the presented framework. Additional sources of information may include other information in bug reports, expert domain knowledge, static code analysis, text mining of source code, etc.

Identifying the set of key-words that contribute to a bug being classified as a potential vulnerability is an important step in better understanding vulnerabilities and further improving classification accuracy. The set of key-words and their relationship may reveal key features of the software package that are more vulnerable. Furthermore, this information can be used to further increase the classification accuracy.

The novel method for dimensionality selection presented in Chapter 5 can be applied to other text mining and non-text mining problems where dimensionality selection is required. The usability and advantages of applying the method to other dimensionality selection problems will be explored in the future.

A classification methodology that attempts to alleviate classification biases of ML based classifiers due to sample size, dimensionality and classifier type will be developed. The methodology relies on data-driven aggregation of results from multiple classifiers trained from multiple sources.

Finally, the knowledge gathered from identifying vulnerabilities can be used to increase the robustness and security of future software releases. Identifying what types of vulnerabilities are most common and where most vulnerabilities occur can lead to development teams taking pre-emptive action against potential future vulnerabilities. Similarly, such information may help future distributions of software to be less vulnerable. Furthermore, by examining bugs that are later identified as vulnerabilities it may be possible to identify packages and source code that may later yield more vulnerabilities. Thus, bugs originating from these packages can be given a higher priority.

REFERENCES

[Ahmed 08] M. F. Ahmed, S. S. Gokhale, "Linux Bugs: Life Cycle and Resolution Analysis," in *Proc of Int. Conf. on Quality Software (QSIC 08)*, pp. 396-401, Aug. 2008.

[Ahmed 09] M. F. Ahmed, S. S. Gokhale, "Linux Bugs: Life Cycle, Resolution and Architectural Analysis," in I*nformation Software Technology*, vol. 5, no. 11, pp. 1618-1627, Nov. 2009.

[Alhazmi 07] O. H. Alhazmi, Y. K. Malaiya, I. Ray "Measuring, analyzing and predicting security vulnerabilities in software systems," in *Computers & Security*, vol. 26, no. 3, pp. 219-228, May 2007.

[Anvik 06] J. Anvik, L. Hiew, G. C. Murphy, "Who Should Fix This Bug?" in *Proc. of Int. Conf. on Software Engineering (ICSE 06)*, pp. 361-370, May 2006.

[Arnold 09] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, A. Kaseorg, "Security Impact Ratings Considered Harmful," in *Proc. of Conf. on Hot Topics in Operating Systems, USENIX*, May 2009.

[Austin 11] A. Austin, L. Williams "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in *Proc. of Int. Symp. on Empirical Software Engineering and Measurement (ESEM 11)*, pp. 97-106, Sep. 2011.

[Axelsson 00] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," in *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 3, pp. 186–205, Aug. 2000.

[Barber 12] D. Barber, *Bayesian Reasoning and Machine Learning*, Cambridge University Press, New York, 2012.

[Basiri 09] M. E. Basiri, S. Nemati, "A novel hybrid ACO-GA algorithm for text feature selection,"" in *Proc. of IEEE Congress on Evolutionary Computation*, pp. 2561-2568, May 2009.

[Battiti 94] R. Battiti, "Using mutual information for selecting features in supervised neural net learning," in *IEEE Trans. on Neural Networks*, vol. 5, no. 4, pp. 537-550, Jul 1994.

[Bell 11] R. M. Bell, T. J. Ostrand, E. J. Weyuker, “Does Measuring Code Change Improve Fault Prediction?,” in *Proc of Promise, Int. Conf. on Predictive Models in Software Engineering*, 2011.

[Cotroneo 12] D. Cotroneo, R. Natella, R. Pietrantuono, “Predicting aging-related bugs using software complexity metrics,” in *Performance Evaluation*, vol. 70, no. 3, pp. 163-176, 2012.

[Cubranic 04] D. Cubranic, G. C. Murphy, “Automatic bug triage using text categorization,” in Proc. of *Int. Conf. on Software Engineering and Knowledge Engineering*, pp. 92-97, Jun. 2004.

[Espejo 10] P. G. Espejo, S. Ventura, F. Herrera, “A Survey on the Application of Genetic Programming to Classification,” in *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 40, no. 2, pp. 121-144, Mar 2010.

[Fellbaum 98] C. Fellbaum, *WordNet: An Electronic Lexical Database*, MIT Press, Cambridge, MA, 1998.

[Goldberg 89] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Professional, 1989.

[Hartmann 82] C. R. P. Hartmann, P. K. Varshney, K. G. Mehrotra, C. Gerberich, “Application of information theory to the construction of efficient decision trees,” in *IEEE Trans. on Information Theory*, vol. 28, no. 4, pp. 565-577, Jul. 1982.

[Ingersoll 13] G. S. Ingersoll, T. S. Morton, A. L. Farris, *Taming Text: How to Find, Organize and Manipulate it*, Manning Publications, New York, 2013.

[Jain 97] A. Jain, D. Zongker, “Feature selection: evaluation, application, and small sample performance,” in *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 19, no. 2, pp. 153-158, Feb 1997.

[Jeong 09] G. Jeong, S. Kim, T. Zimmermann, “Improving bug triage with bug tossing graphs,” in *Proc. of Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT*, pp. 111–120, Aug. 2009.

[Kester 10] D. Kester, M. Mwebesa, J. S. Bradbury, “How Good is Static Analysis at Finding Concurrency Bugs?” in *Proc of IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 10)*, pp. 115-124, Sep. 2010.

[Khoo 10] W. M. Khoo, S. Aloteibi, R. Anderson, M. Meeks, “Hunting for vulnerabilities in large software: the OpenOffice suite,” Cambridge University press, Jun. 2010.

[Ko 06] A. J. Ko, B. A. Myers, D. H. Chau, "A Linguistic Analysis of How People Describe Software Problems," in *Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC 06)*, pp. 127-134, Sep. 2006.

[Kratkiewicz 05] K. Kratkiewicz, R. Lippmann, "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools," in *Proc of Workshop on the Evaluation of Software Defect Detection Tools*, pp. 62-71, Jun. 2005.

[Krsul 98] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue, May 1998. [Online]. Available: http://ftp.cerias.purdue.edu/pub/papers/ivan-krsul/krsul-phd-thesis.pdf

[Lamkanfi 10] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, "Predicting the severity of a reported bug," in *Proc. of IEEE Working Conf. on Mining Software Repositories (MSR 10)*, pp. 1-10, May 2010.

[Lamkanfi 11] A. Lamkanfi, S. Demeyer, Q. D. Soetens, T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," in *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR)*, pp. 249-258, Mar. 2011.

[Lamkanfi 11] A. Lamkanfi, S. Demeyer, Q. D. Soetens, T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," in *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR)*, pp. 249-258, Mar. 2011.

[Li 10] P. Li, B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *Proc. of IEEE Int. Conf. on Information Theory and Information Security (ICITIS)*, pp. 521-524, Dec. 2010.

[Li 11] L. Li, H. Leung, "Mining Static Code Metrics for a Robust Prediction of Software Defect-Proneness," in *Proc. of Int. Symp. on Empirical Software Engineering and Measurement (ESEM 11)*, pp. 207-214, Sep. 2011.

[Liu 98] H. Liu, H. Motoda, "Feature Extraction, Construction and Selection: A Data Mining Perspectiv," in *The Springer International Series in Engineering and Computer Science Series*, vol. 453, 1998.

[Linda 14] O. Linda, D. Wijayasekara, M. Manic, M. McQueen, "Optimal Placement of Phasor Measurement Units in Power Grids Using Memetic

Algorithms," *in Proc. IEEE Int. Symp. on Industrial Electronics*, (Accepted for publication), Jun. 2014.

[MITRE 14] MITRE Corporation (Mar. 2014), Common Vulnerabilities and Exposures (CVE) [Online]. Available: http://cve.mitre.org/.

[Muharram 05] M. Muharram, G. D. Smith, "Evolutionary constructive induction," in *IEEE Trans. on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1518–1528, Nov. 2005.

[MySQL 14] MySQL Bugs (Mar. 2014) [Online] http://bugs.mysql.com/

[Neshatian 08] K. Neshatian, M. Zhang, "Genetic programming and class-wise orthogonal transformation for dimension reduction in classification problems," in Proc. of *European Conference on Genetic Programming*, Mar. 2008.

[Neuhaus 07] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, "Predicting Vulnerable Software Components," in *Proc. of ACM conference on Computer and communications security*, pp. 529-540, 2007.

[Neuhaus 09] S. Neuhaus, T. Zimmermann, "The Beauty and the Beast: Vulnerabilities in Red Hat's Packages," in *Proc. of the 2009 USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[Noll 11] J. Noll, S. Beecham, D. Seichter, "A Qualitative Study of Open Source Software Development: the OpenEMR Project," in Proc. of *Int. Symp. on Empirical Software Engineering and Measurement (ESEM 11)*, pp. 30-39, Sep. 2011.

[Otero 03] F. E. B. Otero, M. M. S. Silva, A. A. Freitas, J. C. Nievola, "Genetic Programming for Attribute Construction in Data Mining," in *Proc. of European Conference on Genetic Programming*, pp. 384-393, Apr 2003.

[Ozment 07] A. Ozment, "Vulnerability discovery and software security," Ph.D. dissertation, University of Cambridge Computer Laboratory, 2007.

[Porter 80] M. F. Porter, "An algorithm for suffix stripping," in *Program*, vol. 14, no. 3, pp. 130-137, 1980.

[Prifti 11] T. Prifti, S. Banerjee, B. Cukic, "Detecting Bug Duplicate Reports through Local References," in Proc of *Int. Conf. on Predictive Models in Software Engineering (PROMISE 11)*, pp. 8:1-8:9, Sep. 2011.

[Quinlan 93] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, 1993.

[Raymer 00] M. L. Raymer, W. L. Punch, E. D. Goodman, L. A. Kuhn, A. K. Jain, "Dimensionality reduction using genetic algorithms," in *IEEE Trans. on Evolutionary Computation*, vol. 4, no. 2, pp. 164-171, Jul 2000.

[Redhat 14] Redhat, Inc. (Mar. 2014), Redhat Bugzilla Main Page [Online]. Available: https://bugzilla.redhat.com/.

[Runeson 07] P. Runeson, M. Alexandersson, O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in Proc. of *Int. Conf. on Software Engineering (ICSE 2007)*, pp. 499-510, May 2007.

[Safavian 91] S. R. Safavian, D. Landgrebe, "A survey of decision tree classifier methodology," in *IEEE Trans. on Systems, Man and Cybernetics*, vol. 21, no. 3, pp. 660-674, May/Jun. 1991.

[Schumacher 00] M. Schumacher, C. Haul, M. Hurler, A. Buchmann, "Data Mining in Vulnerability Databases," in *Proc of 7th Workshop Sicherheit in vernetzten Systemen*, Mar. 2000.

[Sebastiani 02] F. Sebastiani, C. N. D. Ricerche, "Machine learning in automated text categorization," in *ACM Computing Surveys*, vol. 34, pp. 1-47, 2002.

[Simon 13] D. Simon, *Evolutionary Optimization Algorithms - Biologically Inspired and Population-Based Approaches to Computer Intelligence*, Wiley, New Jersey, 2013.

[Shahmehri 12] N. Shahmehri, A. Mammar, E. Montes de Oca, D. Byers, A. Cavalli, S. Ardi, W. Jimenez, "An advanced approach for modeling and detecting software vulnerabilities," in *Information and Software Technology*, vol. 54, pp. 997-1013, 2012.

[Shannon 48] C. E. Shannon, "A Mathematical Theory of Communication," in *Bell System Technical Journal*, vol. 27, pp. 379-423 & 623-656, Jul./Oct. 1948.

[Torri 10] L. Torri, G. Fachini, L. Steinfeld, V. Camara, L. Carro, É. Cota, "An Evaluation of Free/Open Source Static Analysis Tools Applied to Embedded Software," in *Proc of Latin American Test Workshop (LATW 10)*, pp. 1-6, Mar. 2010.

[Uguz 11] H. Uguz, "A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm," in *Knowledge-Based Systems*, vol. 24, no. 7, pp. 1024-1032, Oct. 2011.

[Venter 04] H. S. Venter, J. H. P. Eloff, “Vulnerability forecasting a conceptual model,” in *Computers & Security*, vol. 23, no. 6, pp 489-497, Sep. 2004.

[Wang 08] X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun, “An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information,” *in Proc. of Int. Conf. on Software Engineering (ICSE 08)*, pp. 461-470, May 2008.

[Wen 07] J. Wen; Z. Li, “Semantic Smoothing the Multinomial Naive Bayes for Biomedical Literature Classification,” in *Proc. IEEE Int. Conf. on Granular Computing*, pp. 648-648, Nov. 2007.

[Wenke 01] L. Wenke X. Dong, “Information-theoretic measures for anomaly detection,” in *Proc. of IEEE Symp. on Security and Privacy*, pp. 130-143, 2001.

[Wijayasekara 11] D. Wijayasekara, M. Manic, P. Sabharwall, V. Utgikar, “Optimal artificial neural network architecture selection for performance prediction of compact heat exchanger with the EBaLM-OTR technique,” in *Nuclear Engineering and Design*, vol. 241, no. 7, pp. 2549–2557, July 2011.

[Wijayasekara 12] D. Wijayasekara, M. Manic, J. L. Wright, M. McQueen “Mining Bug Databases for Unidentified Software Vulnerabilities,” in *Proc. of Intl. IEEE Intl. Conference on Human System Interaction (HSI)*, Jun. 2012.

[Wijayasekara 13] D. Wijayasekara, M. Manic, M. McQueen, “Information Gain Based Dimensionality Selection for Classifying Text Documents,” in *Proc. of IEEE Congress on Evolutionary Computation (IEEE CEC)*, Jun. 2013.

[Wright 13] J. L. Wright, J. W. Larsen, M. McQueen, “Estimating Software Vulnerabilities: A Case Study Based on the Misclassification of Bugs in MySQL Server,” in *Proc. of Int. Conf. on Availability, Reliability and Security (ARES)*, pp. 72-81, Sep. 2013.

[Wu 11] L. Wu, B. Xie, G. Kaiser, R. Passonneau, “BugMiner: Software Reliability Analysis Via Data Mining of Bug Reports,” in *Proc. of Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, pp. 95-100, Jul. 2011.

[Yamaguchi 11] F. Yamaguchi, F. 'FX' Lindner, K. Rieck, “Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning,” in *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2011.

[Yang 97] Y. Yang, J. O. Pedersen, "A comparative study on feature selection in text categorization," in *Proc. of Int. Conf. on Machine Learning*, pp. 412–420, 1997.

[Zhanga 12] R. Zhanga, S. Huanga, Z. Qi, H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," in *Computers and Mathematics with Applications*, vol. 63, pp. 469-480, 2012.

[Zitser 04] M. Zitser, R. Lippmann, T. Leek "Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code," in *Proc. of Int. Symp. on Foundations of Software Engineering (FSE 04), ACM SIGSOFT*, pp. 97–106, Nov. 2004.

APPENDIX A – LIST OF PUBLICATIONS

This appendix presents an overview of the author's published or submitted journal and peer-reviewed conference publications.

Journal Publications

[1] D. Wijayasekara, M. Manic, P. Sabharwall, V. Utgikar, "Optimal artificial neural network architecture selection for performance prediction of compact heat exchanger with the EBaLM-OTR technique," in *Nuclear Engineering and Design*, vol. 241, no. 7, pp. 2549–2557, July 2011.

Abstract: Artificial Neural Networks (ANN) have been used in the past to predict the performance of printed circuit heat exchangers (PCHE) with satisfactory accuracy. Typically published literature has focused on optimizing ANN using a training dataset to train the network and a testing dataset to evaluate it. Although this may produce outputs that agree with experimental results, there is a risk of over-training or overlearning the network rather than generalizing it, which should be the ultimate goal. An over-trained network is able to produce good results with the training dataset but fails when new datasets with subtle changes are introduced. In this paper we present EBaLM-OTR (error back propagation and Levenberg-Marquardt algorithms for over training resilience) technique, which is based on a previously discussed method of selecting neural network architecture that uses a separate validation set to evaluate different network architectures based on mean square error (MSE), and standard deviation of MSE. The method uses k-fold cross validation. Therefore in order to select the optimal architecture for the problem, the dataset is divided into three parts which are used to train, validate and test each network architecture. Then each architecture is evaluated according to their generalization capability and capability to conform to original data. The method proved to be a comprehensive tool in identifying the weaknesses and advantages of different network architectures. The method also highlighted the fact that the architecture with the lowest training error is not always the most generalized and therefore not the optimal. Using the method the testing error achieved was in the order of magnitude of within 10−5–10−3. It was also show that the absolute error achieved by EBaLM-OTR was an order of magnitude better than the lowest error achieved by EBaLM-THP.

Peer-Reviewed Conference Publications

[2] D. Wijayasekara, M. Manic, J. L. Wright, M. McQueen "Mining Bug Databases for Unidentified Software Vulnerabilities," in *Proc. of Intl. IEEE Intl. Conference on Human System Interaction (HSI)*, Jun. 2012.

Abstract: Identifying software vulnerabilities is becoming more important as critical and sensitive systems increasingly rely on complex software systems. It has been suggested in previous work that some bugs are only identified as vulnerabilities long after the bug has been made public. These vulnerabilities are known as hidden impact vulnerabilities. This paper discusses existing bug data mining classifiers and present an analysis of vulnerability databases showing the necessity to mine common publicly available bug databases for hidden impact vulnerabilities. We present a vulnerability analysis from January 2006 to April 2011 for two well known software packages: Linux kernel and MySQL. We show that 32% (Linux) and 62% (MySQL) of vulnerabilities discovered in this time period were hidden impact vulnerabilities. We also show that the percentage of hidden impact vulnerabilities in the last two years has increased by 53% for Linux and 10% for MySQL. We then propose a hidden impact vulnerability identification methodology based on text mining classifier for bug databases. Finally, we discuss potential challenges faced by a development team when using such a classifier.

[3] D. Wijayasekara, M. Manic, M. McQueen, "Information Gain Based Dimensionality Selection for Classifying Text Documents," in *Proc. of IEEE Congress on Evolutionary Computation (IEEE CEC)*, Jun. 2013.

Abstract: Selecting the optimal dimensions for various knowledge extraction applications is an essential component of data mining. Dimensionality selection techniques are utilized in classification applications to increase the classification accuracy and reduce the computational complexity. In text classification, where the dimensionality of the dataset is extremely high, dimensionality selection is even more important. This paper presents a novel, genetic algorithm based methodology, for dimensionality selection in text mining applications that utilizes information gain. The presented methodology uses information gain of each dimension to change the mutation probability of chromosomes dynamically. Since the information gain is calculated a priori, the computational complexity is not affected. The presented method was tested on a specific text classification problem and compared with conventional genetic algorithm based dimensionality selection. The results show an improvement of 3% in the true positives and 1.6% in the true negatives over conventional dimensionality selection methods.