Empirical Analysis and Automated Classification of Security Bug Reports

Jacob P. Tyo

Thesis submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Electrical Engineering

Katerina Goseva-Popstojanova, Ph.D., Chair
Roy S. Nutter, Ph.D.
Matthew C. Valenti, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2016

Keywords: Cybersecurity, Machine Learning, Issue Tracking Systems, Text Mining, Text Classification

Copyright 2016 Jacob P. Tyo

Abstract

Empirical Analysis and Automated Classification of Security Bug Reports

Jacob P. Tyo

With the ever expanding amount of sensitive data being placed into computer systems, the need for effective cybersecurity is of utmost importance. However, there is a shortage of detailed empirical studies of security vulnerabilities from which cybersecurity metrics and best practices could be determined. This thesis has two main research goals: (1) to explore the distribution and characteristics of security vulnerabilities based on the information provided in bug tracking systems and (2) to develop data analytics approaches for automatic classification of bug reports as security or non-security related. This work is based on using three NASA datasets as case studies. The empirical analysis showed that the majority of software vulnerabilities belong only to a small number of types. Addressing these types of vulnerabilities will consequently lead to cost efficient improvement of software security. Since this analysis requires labeling of each bug report in the bug tracking system, we explored using machine learning to automate the classification of each bug report as a security or non-security related (two-class classification), as well as each security related bug report as specific security type (multiclass classification). In addition to using supervised machine learning algorithms, a novel unsupervised machine learning approach is proposed. An accuracy of 92%, recall of 96%, precision of 92%, probability of false alarm of 4%, F-Score of 81% and G-Score of 90% were the best results achieved during two-class classification. Furthermore, an accuracy of 80%, recall of 80%, precision of 94%, and F-Score of 85% were the best results achieved during multiclass classification.

Acknowledgments

This work was funded by the NASA Software Assurance Research Program (SARP) grant in FY16. I would like to thank Dr. Katerina Goseva-Popstajanova for her support, guidance, and patience throughout this research, as well as my committee members Dr. Roy Nutter and Dr. Matthew Valenti for sharing their knowledge and time with me. Furthermore, I would like to express my sincere gratitude to my parents, siblings, and friends for their unwavering support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The awareness of hacking has risen significantly in recent years due to both an increase in the amount of hacking related media available, and to the seemingly endless attacks plaguing news headlines. The consequences of falling victim to one of these increasing number of attacks has caused a significant shift in software validation, verification, and security monitoring. Security is increasingly becoming part of the software development life cycle. This thesis has two main research goals: (1) to explore the distribution and characteristics of security vulnerabilities based on the information provided in bug tracking systems and (2) to develop data analytics approaches for automatic classification of bug reports as security or non-security related.

This work uses three NASA datasets extracted from issue tracking systems. Two of the datasets used in this thesis are from the same NASA flight mission, with one originating from the developers of that system and the other originating from the IV&V analysts. The remaining dataset originates from an IV&V analysis of a NASA ground mission.

Chapter 3 of this thesis focuses on analyzing the data extracted from bug tracking systems to develop vulnerability profiles, which provide an empirical view of a software security vulnerabilities. This allows for analysts and developers to examine the most common security flaws, as well as the most vulnerable components and subsystems found in their systems. Furthermore, this information can be used to enhance the education of the developers of these systems to eliminate similar issues in the future.

Creating a vulnerability profile of a project using information extracted from a bug track-

ing system requires that each issue is labeled with a tag describing it's security effects, flaws, or type. This information was partially present in one dataset, and non-existent in the others. Assigning these labels is very labor intensive, time consuming, and requires significant security specific knowledge. To alleviate this burden, Chapter 4 focuses on developing machine learning approaches capable of automatically assigning each bug report with its corresponding security tag. This includes classifying each bug report as security related or not, as well as classifying each issue as a specific vulnerability type.

Supervised approaches to automating the classification of issues requires manual labeling which is time consuming and costly. In an attempt to avoid this requirement, a novel unsupervised learning approach was developed.

1.1 Key Terms

This thesis relies on terminology specific to the security field, as well as defines some terms to simplify certain explanations. These terms are addressed in this section:

A **failure** is a departure of the system or system component behavior from its required behavior.

A **fault** is an accidental condition, which if encountered, may cause the system or system component to fail to preform as required. **Bug** is typically used as a synonym for fault.

Bug reports (also referred to as **issues**) are accounts of faults and failures. Bug reports can cover an extreme variety of topics, and are used to detail, find, and fix such system problems.

Developer Change Requests (DCRs) are requests to change or update the system's functionality, or to request a fix for software bugs. These are located in any bug tracking system originating from developers. For the NASA mission containing such requests, each DCR was labeled with a subtype such as "Defect" or "Change Request" which allows for extraction of only DCRs that are bug reports.

Bug tracking systems (sometimes referred to as **issue tracking systems** are software programs that manage and maintain lists of issues as needed by an organization. Many types of these systems exists, yet the type of system is irrelevant for this research.

A **vulnerability** (also referred to as a **security issue**) is a security flaw, glitch, or weakness found in software that can lead to gaps in the systems security, and potentially be exploited by an attacker.

A **vulnerability profile** is an empirical view of software security vulnerabilities. The vulnerability profile of each project includes the most common types of vulnerabilities in the system, the most vulnerable components, where most vulnerabilities are found, and more.

CWE refers to Common Weakness Enumeration which is a formal list of software weakness types aimed at serving as a common language for describing software security weaknesses in architecture, design, or code. This work will refer both to the total listing (CWE list) as well as individual elements within the list (CWE).

A **CWE View** is a particular perspective (or view) from which to look at the CWE list. Each view can organize, categorize, or group each individual CWE's within the overall list differently.

CVE stands for Common Vulnerabilities and Exposures. CVE is a dictionary of common names for publicly known cyber security vulnerabilities and exposures.

NVD (National Vulnerability Database) is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol. NVD includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics.

CVSS is the Common Vulnerability Scoring System which is an open framework for communicating the characteristics and severity of software vulnerabilities. This scoring system takes into account properties of vulnerabilities such as the impact it would have if exploited, as well as the ease of exploiting the vulnerability.

1.2 Research Questions and Contributions

This work explores the following research questions:

1. What are the dominate types of vulnerabilities in NASA ground and flight software systems?

   (a) Are they consistent across projects and project types?

2. Can supervised machine learning algorithms be used to classify software issues as security related or not?

   (a) Do some learners preform consistently better than others?

   (b) How much data must be set aside for training in order to produce accurate classification results?

3. Can supervised machine learning algorithms be used to classify security issues to specific security classes?

   (a) Are some classes harder to predict than others?

4. Can unsupervised machine learning algorithms be used to classify software issues as security related or not?

5. How does the performance of supervised and unsupervised machine learning algorithms compare when classifying software bug reports?

The main contributions of this thesis are as follows:

We developed a vulnerability profile for thee NASA datasets, which include over 1,800 bug reports. The most common security flaws in each dataset were determined, and those which were consistent across the different datasets were identified. Software faults and failures in open source and NASA missions have been previously studied [1], [2], [3], but not specifically vulnerabilities.

Supervised machine learning algorithms were used to classify each bug report as security related or not (two-class problem), based only on the information found in a bug tracking system. Several efforts have been done [4], [5], [6], and [7], which focused on separating security from non-security bug reports: [4], [5], and [6] automated this process achieving moderate performance. Furthermore, all approaches were only focused on classification to security and non-security bug reports.

Supervised machine learning algorithms were used to classify each bug report to a specific security class (multiclass classification), based only on the information found in a bug tracking system. Wang et al used machine learning techniques to classify vulnerabilities as a CVE type [8]. However, the features used for classification were the CVSS scores of the vulnerability.

An unsupervised machine learning approach was developed to classify bug reports as security related or not. This approach uses a one class problem and incorporating the CWE list. It appears that no prior work using unsupervised machine learning to classify software security bug reports exists.

Chapter 2

Related Work

This chapter presents the related work to the vulnerability profile and the automated security bug classification system.

2.1 Vulnerability Profile

The information housed within an issue tracking system contains valuable information for quantifying traits of a software system such as how error prone it is, how well up kept it is, and how secure the system is. Empirical studies of the information found in issue tracking systems can be used for things ranging from determining the most error prone aspects of a system to detecting how the system could be improved. While little research which leverages an issue tracking system in an empirical way was found, the following papers are highly related to such a situation.

Fenton et al empirically studied a large software-intensive telecommunication application from Ericsson Telecom AB [9]. This work was focused on a range of software engineering hypotheses related to the Pareto principle of distribution of faults and failures, the use of early fault data to predict later fault and failure data, metrics for fault prediction, and benchmarking fault data. The results included strong evidence that a small number of modules contain most of the faults discovered in prerelease testing and that a very small number of modules contain most of the faults discovered in Operation. However, the fault prone modules was not explained by it's size or complexity. While showing that the number

of faults discovered in prerelease testing is an order of magnitude greater than the number discovered in 12 months of operational use, they also discovered fairly stable numbers of faults discovered at corresonding testing phases. The most surprising result was related to the a counter-intuitive relationship between pre- and postrelease faults. In other words, the modules which were the most fault-prone prerelease were among the the lease fault-prone postrelease, as well as vice versa.

Hamill et al explored the localization of faults that lead to individual software failures and the distribution of different types of software faults in a mature NASA mission and the C preprocessor of GCC [3]. It was shown that individual failures are often caused by multiple faults spread throughout the system, which indicates that finding and fixing faults that lead to such software failures in large, complex systems are often difficult and challenging tasks despite the advances in software development. While showing that the most common types of software faults are requirement faults, coding faults, and data problems, they also showed that a significant percentage of failures are linked to late life cycle activities contrary to popular belief. Comparing the trends across software systems, Hamill et al suggested that these trends were likely to be intrinsic characteristics of software faults and failures rather than project specific.

Hamill et al focused on empirically characterizing software fixes [2] based on their previous work [3]. Based on a safety-critical NASA mission containing 21 large-scale software components, a link was established from software faults to failures and consequently to fixes made to correct these faults. The results showed that a significant number of software failures required fixes in multiple software components and/or multiple software artifacts (i.e.,15% and 26% respectively). Furthermore, not only did the patterns of software components that were often fixed together were significantly affected by the software architecture, but also the types of fixed software artifacts were highly correlated with fault type and they had different distributions for pre-release and post-release failures.

Hamill et al studied the types of faults that caused software failures, activities taking place when faults were detected or failures were reported, and the severity of failures [1] which builds on their previous works [2] and [3]. They explored the associations among these attributes and the trends within and across releases. Results showed that only a

few fault types were responsible for the majority of failures pre-release, post-release, and across releases. The distributions of fault types were different for pre-release and post-release failures, and the percentage of safety-critical failures was small overall. All failures were more heavily associated with coding faults than with any other type of fault, while components that experienced a high number of failures in one release were not necessarily among high failure components in the subsequent release. Lastly, components that experienced more failures pre-release were more likely to fail post-release.

With the increasing popularity, capability, and the release of open source OSes for hand-held devices, Maji et al utilized bug reports, bug fixes, developer reports, and failure reports to look into the manifestation of failures in different modules of Android and their characteristics [10]. These reports were viewed from the standpoint of the frequency of failures, and the persistence of the issue. This study showed that most of the bugs (over 90%) were permanent in nature, the kernel layer was sufficiently robust yet much effort was needed to improve the middleware layer. Furthermore, between 11% and 50% of bugs were a result of the customizability Android offered to the developers, and most bugs required only minor changes for correction (etc. update configuration parameters).

Grottke et al [11] analyzed the faults discovered in the on-board software for 18 JPL/NASA space missions, pointing out that the ability to effectively deal with faults is increasingly important as space mission software becomes more complex. The authors defined Bohrbugs as bugs that are easily isolated and removed during software testing, and Mandelbugs as bugs that appear to behave chaotically. This paper explored the proportions of Bohrbugs and Mandelbugs and studies how they evolved over time. Furthermore, they examined whether or not the fault type and attributes such as the failure effect are independent. A set of 520 anomalies was derived, each of which represent a unique fault in the flight software of 18 JPL/NASA missions. Grottke et al determined that there is a highly significant relationship between the fault type and the failure risk, and 61.4% of bugs were Bohrbugs, and 36.5% were Mandelbugs.

Frattini et al presented an analysis of 146 bug reports from Apache Virtual Computing Lab, which they determined to be a representative open source Cloud platform [12]. This analysis identified the components where bugs were likely to be found in future releases,

the phases of the service life cycle during which such bugs may be discovered, and the modification required to solve them. This paper was based on a small dataset, but the results included useful information which could be used to create guidelines and increase efforts in areas in which the system under concern is weakest.

Alonso et al [13] analyzed the mitigation's associated with each fault defined in their previous work [11], [14]. Trends of mitigation type proportions within missions as well as from mission to mission were identified, while looking for relationships between fault types and mitigation types. This paper showed that the Bohrbugs and Mandelbugs discussed in [11] are most frequently mitigated via fixes instead of other measures such as proactive reboots, and for each type of fault, the earlier missions tended to show lower frequencies of fixes/patches than the more recent missions.

Xia et al utilized the bug databases and code repositories for the build systems Ant, Maven, CMake, and QMake containing 199, 250, 200, and 151 bug reports respectively [15]. Each sample was manually classified into various categories for further analysis. These categories were very general. The results showed that 21.35% of bugs belonged to the external interface category, 18.23% belonged to the logic category, and 12.86% belonged to the configuration category.

Alhazmi et al examined the feasibility of quantitatively characterizing some aspects of security [16]. Specifically, they investigated if it is possible to predict the number of vulnerabilities that can potentially be present in a software system, but may not have been found yet. The density of vulnerabilities, fraction of software defects that are security related, the dynamics of vulnerability discovery, and the vulnerability discovery rate were used to estimate the magnitude of the undiscovered vulnerabilities still present in the system. The analysis was based on both commercial and open-source systems to determine the generalizability. The results revealed that the vulnerability densities fall within a range of values, similar to fault density for general faults. The authors claimed that it is possible to model the vulnerability discovery using a logistic model, which can sometimes be approximated by a linear model as a function of time.

Venter et al focused on the problem in which each vulnerability scanner represents, identifies, and classifies vulnerabilities in its own way [17]. A vulnerability scanner is a proactive

information security technology which searches systems and networks for the occurrence of known flaws and then produces a report that an individual or Enterprise can use to strengthen its security. A static code analysis tool is a type of vulnerability scanner. Each scanner's report is different, therefore scanners are difficult to compare. Often times multiple scanners are ran on the same system to catch as many vulnerabiities as possible, yet it is difficult to aggregate the results appropriately. This paper outlines an approach towards achieving a standardized vulnerability category set via a data-clustering algorithm, which was done with self-organizing maps (SOMs) and data pulled from CVE's. While no quantification of the results was provided, this paper showed the importance and benefits of having a standardized vulnerability categorization set.

The Sourcefire Vulnerability Research Team (Younan et al) took a historical look at vulnerabilities reported from 1998 to 2012 [18]. This vulnerability analysis was based on frequency analysis of the CVE's in the NVD databases. Younan et al were able to leverage this data to draw several interesting conclusions, one of which being that despite the progress in mitigating attacks against buffer overflows, they remain one of the top ranking vulnerabilities year over year. Furthermore, they showed that while fewer vulnerabilities were reported in the last couple of years, the percentage of more critical vulnerabilities has increased. Some surprising conclusions include that Microsoft has significantly improved within the last couple of years and their browser and mobile operating systems are better than their competitors' in terms of vulnerabilities discovered. Furthermore, Chrome ranked as one of the highest for vulnerabilities, while Android had very few; iPhone had a significant lead in number of vulnerabilities, while Safari had the fewest compared to the other browsers.

2.2 Automated Bug Report Classification

Issue tracking systems contain unstructured text, and therefore text mining can be used to automatically process data from such systems. This section discusses previous work that uses the issues available in issue tracking systems to perform some type of automatic classification.

Hovsepyan et al approached static code analysis from the perspective of raw text, i.e. treated the source code as a text document [19]. Their text mining and classification approach used consisted of creating the feature vectors which contained the term frequencies and Support Vector Machine (SVM) to classify which files contain vulnerabilities. The dataset used in this work was the source code for the K9 mail client for Android mobile device applications, and labeled with Fortify [20]. Fortify is a static code analysis tool which does not detect 100% of vulnerabilities and is known to have a very high false positive rate. Because this was used to label the data, the performance metrics reported are compared to these labels and not the true class of the data meaning that the reported accuracy of 0.87, precision of 0.85 and recall of 0.88 mean almost nothing in terms of true performance, only that this method performs similar to Fortify.

Scandariato et al took an approach very similar to that mentioned in [19] by also using project source code as text for text mining approaches [21]. Specifically, they aimed to determine which components of a project are likely to contain vulnerabilities using term frequencies along with either Naive Bayes or Random Forest. After validation with 20 Android application, they determined that a dependable prediction model can be built (precision between 62% and 100% and recall between 48% and 100%), which could be useful in prioritizing the validation activities.

Perl et al approached the increasing level of software vulnerabilities from a perspective yet to be mentioned, through Vulnerability Contributing Commits (VCC) within version control systems [22]. A large scale mapping was created between CVE's and the commits leading to them to create a vulnerable commit database. Based on that database, an SVM classifier was used to flag suspicious commits. 66 projects that used either C or C++ programming language as well as the Git version control system were used for the analysis. During testing, the danger level of a piece of code was determined by the danger level of the commit. The authors stated that compared to Flawfinder [23], their method cut the number of false positives in half, while maintaining a recall between 26% and 48% and precision between 11% and 56%.

Jalbert et al [24] proposed a system that automatically classifies duplicate bug reports as they arrive, using surface features, textual semantics, and graph clustering. Titles and

descriptions were used from the bug reports, however the authors argued that combining them would result in a loss of information and are therefore kept as separate corpora. A "bag of words" approach was used when defining similarly between the textual data, or in other words the cosine similarity was computed between the term frequency vectors of each issue. To account for weighting of features, it is argued that the popular Term Frequency / Inverse Document Frequency (TF/IDF) would not provide effective results, therefore they developed their own logarithmic weight algorithm based solely on the number of occurrences of a term in a document. The dataset consisted of 29,000 bug reports from the Mozilla project. The results showed that the created system was capable of filtering out only a small portion (8%) of duplicate bug reports.

Lamkanfi et al demonstrated that text mining can predict the severity of a given bug report with a reasonable accuracy given a training set of sufficient size [25]. In this paper four well-known text mining algorithms were compared (Naive Bayes, Naive Bayes Multinomial, K-Nearest Neighbor and Support Vector Machine). Three research questions were explored: "What classification algorithm should we use when predicting bug report severity?", "How many bug reports are necessary when training a classifier in order to have reliable predictions?", and "What can be deduced from the resulting classification algorithms?" The results showed that the Naive Bayes Multinomial classifier had the best accuracy (as measured by ROC) and was also the fastest when classifying the severity of reported bugs. The Naive Bayes and Naive Bayes Multinomial classifiers are able to achieve stable accuracy the fastest, needing about 250 bugs for training.

Antoniol et al investigated whether the text of the issues posted in bug tracking systems is enough to classify them into corrective maintenance and other kinds of activities [26]. The Eclipse, Mozilla, and JBoss open soure systems were the selected datasets, with the title, description and discussing being used to build the feature vector. Although a lot of focus was placed on automating this process, it was also shown that the information contained in issues posted on bug tracking system can be indeed used to classify such issues, distinguishing bugs from other activities, with a precision between 64% and 98% and a recall between 33% and 97% and accuracy as high as 82%.

Chawla et al attempted to classify an issue as either a bug or other request using a fuzzy

logic approach [27]. The typical text preprocessing steps were applied, and three open source software systems were used as case studies: HTTPClient, Jackrabbit, and Lucene. A custom feature extraction process was used, yet it was very similar to the common TF-IDF method, followed by 5 fold cross validation for separation of the data into training and testing sets. The fuzzy logic approach worked marginally better than other classification methods such as Naive Bayes, Logistic Regression, and AD Tree, and achieved accuracies of 87%, 84%, and 91% and recalls of 77%, 78%, and 85% on the three projects respectively.

Ahmed et al pointed out the need for dependable bug categorization in order to better handle proper solution [28]. Their paper analyzes the automatic prediction of different bug types (Function, Logic, Standard, and GUI) using K Nearest Neighbor and Naive Bayes. After using 10 fold cross validation, they report recall and precision respectively 91% and 75% for standard issues, 79% and 75% for function issues, 79% and 73% for user interface issues, and 72% an 79% for logic issues respectively.

Somasundaram et al investigated automatically categorizing bug reports to allow for the assignment of a bug to the proper development team through the use of TF-IDF with an SVM classifier (SVM-TF-IDF), LDA with SVM (SVM-LDA), and LDA and KL (LDA-KL, Kullback Leiber divergence) [29]. Kullback Leiver divergence is an approach which classifies bug reports by measuring the divergence between each topic's centroids obtained from LDA and a test bug. After testing on the Eclipse, Mylyn, and Mozilla datasets, LDA-KL produced recalls of 86%, 77%, and 82% on the three project respectively. These results were similar to those found previously but with better consistency across all components for which bugs must be categorized.

Layman et al applied topic modeling to a corpus of NASA problem reports to extract trends in testing and operational failures, where problem reports are records of off-nominal performance, deviations from design, and human errors that occur while building and operating these systems [30]. The analysis of problem reports with topic modeling led to the most popular topics within and across missions, and how popular topics changed over the lifetime of a mission. Layman et al found that topic modeling can identify problem themes within missions and across mission lifetime. However, they identified multiple challenges: the process of selecting the topic modeling parameters lacks definitive guidance, defining

semantically-meaningful topic labels requires non-trivial effort and domain expertise, topic models derived from the combined corpus of missions were biased toward the larger missions, and topics must be semantically distinct as well as cohesive to be useful.

Wang et al proposed a novel model and methodology to classify and categorize vulnerabilities according to their security types [8]. Furthermore, they used Bayesian Networks to automate the proposed process. The security types were defined as a subset of the NVD classification scheme, and each vulnerability was classified as one of these types based on its CVSS Access Vector, Access Complexity, Authentication, Confidentiality Impact, Integrity Impact, and Availability Impact [31]. In order for this method to be successful, the probability distribution of vulnerabilities was calculated from all vulnerabilities in the NVD related to Firefox. No performance metrics were given, but the authors claimed each software product must use its own Bayesian network, which implies that each software project would need its own network constructed to utilize the proposed methodology. Furthermore, the authors claimed that the automatically generated results were compared to the CVE type in NVD, and it "proved the correctness of our method."

Wright et al conducted an experiment to estimate the number of misclassified bugs yet to be identified as vulnerabilities in the MySQL bug report database [7]. To determine which issues were misclassified, a scoring system was developed in which the first part of the scoring system was the creation of a list of strings with an associated weight, then the second part was simply checking each issue for the presence of any of those strings. If an issue contained any of the strings, then the weight associated with the present string was added to that issues score. This experiment was preformed on a subset of issues from the MySQL bug database, and after scoring, the results were extrapolated into the entire dataset. It was claimed that human efforts are largely ineffective in classifying bugs as vulnerabilities, and with the assumption that any issue including one or more of the strings in the generated list is a vulnerability, after extrapolation they estimated a 657% to 772% increase in the number of vulnerabilities for the MySQL project but did not verify these findings.

Gegick et al used text mining on the descriptions of bug reports to train a statistical model on manually-labeled bug reports to identify security bug reports that were mislabeled as non security bug reports [6]. The term by document frequency method was used to create

feature vectors. More specifically, the SAS text mining tool is used for the feature vector creations, as well as prediction in the form of singular value decomposition (SVD). The issue data from four large Cisco projects were used as datasets which the authors stated that their model identifies a high percentage (77%) of the security bug reports which were manually mislabeled as non-security bug reports by bug reporters. However, this system had a very high false positive rate varying from 26.7% to 96.2%.

Often times, bugs are only identified as vulnerabilities long after the bug has been made public [5]. Wijayasekara et al denoted such issues as Hidden Impact Bugs (HIBs), and created a system that can identify such bugs as an extension of their previous work [32]. CVE's for the Linux kernel were identified, and then corresponding issues were gathered. The text mining method used was a basic "bag of words" approach where the frequency of the terms in each issue was placed into a feature vector. A corpus of regular bugs and HIBs were then created for training and testing. The Naive Bayes, Naive Bayes Multinomial, and Decision Tree classifiers were tested, resulting in a precision of 0.88, 0.78, 0.28, a recall of 0.02, 0.09, and 0.40 respectively.

Behl et al published a paper which highly relates to our work [4], however this paper does not seem to be credible. They claim to have used "the bugzilla repository of bug reports," however bugzilla is an issue tracking system software suite and has no repository of bug reports. This work claimed to use Term Frequency-Inverse Document Frequency (TF-IDF) along with an undefined "vector space model," and compared this performance to an approach using Naive Bayes. Accuracy and precision were the only performance metrics used, which do not relate well to the systems ability to classify bugs as security or non-security. Both performance metrics can give misleading results when using imbalanced datasets, which is expected in this situation. The reported accuracy and precision (95.7% and 93.2% respectively) is only marginally better than Naive Bayes.

Chapter 3

Vulnerability Profiles

A vulnerability profile, as defined in Section 1.1, is an empirical view of software security vulnerabilities. Three NASA bug tracking systems were used to create three separate datasets, in which each security issue was labeled into a specific security class. This information was then used to look for trends that could potentially differentiate security from non-security issues, as well as the security themes that each dataset contains the most of. In this chapter, we explore research questions 1 and 1a.

1. What are the dominate types of vulnerabilities in NASA ground and flight software systems?

    a) Are they consistent across projects and project types?

The procedures and results presented in this section are the result of an empirical study based on the data described in detail in the following section. Following the dataset description, several potential classification schema's are described, and the schema selected for use in this work discussed. The approach taken to manually classify (label) each issue is then described, followed by the results for each dataset. This section is concluded by comparing the datasets for common trends and findings, the threats to validity of this empirical study, and finally by answering the aforementioned research questions 1 and 1a.

3.1 Datasets

Three main datasets from NASA used were utilized for this work: ground mission IV&V issues, flight mission IV&V issues, and flight mission developer issues. The developer issues are obtained from the developers of the software whereas the IV&V issues are obtained from the Independent Verification and Validation (IV&V) analysts. The issue reports by the IV&V analysts were of very good quality containing in depth analysis of the problems. The data extracted from the developer's bug tracking system was also good, yet did not contain quite the same level of detail and security specific information. The datasets discussed below were created from all "closed" issues from their corresponding bug tracking systems.

The first dataset is the IV&V issues extracted from the bug tracking system of a NASA ground mission and will be referred to as Ground Mission IV&V Issues. This system consisted of approximately 1.36 million source lines of code, and the bug tracking system contained 1,779 issues created over four years. The IV&V analysts put special emphasis on considering the security impact of each issue, and as a result 350 (20%) of the issues were marked as potentially security related. Most issues contained very detailed descriptions, titles, comments, and the issues determined to be security related were labeled as such. In addition, the issue descriptions contained security related terminology making it a very good dataset for this project. The fields are detailed in Table B.1.

The second dataset is the IV&V issues extracted from the bug tracking system of a NASA flight mission and will be referred to as Flight Mission IV&V Issues. This system consisted of approximately 924 thousand source lines of code, and the bug tracking system contained 506 issues created over four years. After removal of issues marked as "Withdrawn" or "Not an Issue," 383 remained. Although this dataset was also from the IV&V analysts, there was no special consideration put towards the security of each issue. This resulted in the issue descriptions containing very little security related terminology, and were focused mainly on proper system operation. The fields of this bug tracking system are detailed in B.1.

The third dataset is the Developer issues extracted from the bug tracking system of the same NASA flight mission as Flight Mission IV&V Issues and will be referred to as Flight Mission Developer Issues. This bug tracking system consisted of 1,947 Developer Change

Requests (DCRs) created over five and a half years. 573 of these DCRs were marked as "Defects," whereas the others were marked as "Change Requests" or some other non-issue related category. Only "Defect" DCRs were included in the dataset, and any mentioning of an issue from this project refers to only the "Defect" DCRs. This dataset originated from developers instead of the IV&V analysts, resulting in a much greater focus on proper project operation instead of security. The fields of this bug tracking system are detailed in B.2.

The aforementioned datasets originate from one of two sources (IV&V analysts or developers) and from one of two projects (flight mission or ground mission). This creates natural groupings for comparison. The ground mission IV&V issues dataset will be compared to flight mission IV&V issues dataset to identify trends consistent across ground and flight missions. The flight mission IV&V dataset will be compared against the flight mission developer issues to identify problematic themes across IV&V and developer issues.

3.2 Classification Schema

In order to classify a bug report, a classification schema is needed. Common problems among classification schemas are undefined levels of specificity, very complicated structure, and non-hierarchical graphs. This section explores some software vulnerability and/or weakness classification schemes along with some strengths and weaknesses of each. At the end of this section, one of the mentioned classification schemes will be selected for use in the rest of this work.

Common Weakness and Enumeration (CWE) is a formal list of software weakness types aimed at serving as a common language for describing software security weaknesses in architecture, design, or code. The CWE list also serves as a standard measuring stick for software security tools targeting those weaknesses, and to provide a common baseline standard for weakness identification, mitigation, and prevention efforts [33]. Each entry in this list is given a number such as CWE-120, describes a security weakness, and from here on out a CWE will refer to an individual entry (i.e. CWE-120) in the CWE formal list. As quickly getting information from a list of 1004 CWE's is nearly impossible, a number of views (as defined in Section 1.1) have been developed to ease grouping similar CWE's based

on differing factors, as well as easing the use of this general knowledge. Many views will be discussed and compared in this section, which attempt to minimize or eliminate common problems such as undefined levels of specificity, very confusing non-hierarchical structure, or structuring consisting of child CWE's having multiple parent CWE's (as defined in Section 1.1).

Similar to CWE is the Common Vulnerabilities and Exposures (CVE) dictionary [34]. This contains common names for publicly known cybersecurity vulnerabilities and often includes examples, descriptions, and are mapped to one or more CWE's. In more general terms, a CVE is an account of a publicly know vulnerability, which is often grouped into one or more CWE's in order to describe the type of underlying problem more accurately and in more common terms. Because this is a dictionary of known vulnerabilities, it cannot be used as a classification schema, but could possibly be leverages to get real world examples of specific CWE's.

CWE-2000 is the Comprehensive CWE Dictionary View (from here on denoted as CWE-2000) covers all elements in CWE. It contains 1004 total CWE's with no particular grouping or classification [35]. This listing of 1004 CWE's offers no classifying advantage as there is no relationships drawn, similarities defined, or any attempt to ease the confusing nature of classifying such a large corpora of software weaknesses. This view will not be considered as a potential classification schema for use in this project.

CWE-1000 is the Research Concepts view and was created with the intent to facilitate research into weaknesses, including their inter-dependencies and their role in vulnerabilities [36]. It classifies weaknesses in a way that largely ignores how they can be detected, where they appear in code, and when they are introduced in the software development life-cycle. Instead, it is mainly organized according to the abstractions of software behaviors. It uses a deep hierarchical organization which provides many levels of abstraction and specificity. Where possible, this view uses abstractions that do not consider particular languages, framework, technologies, life-cycle development phases, frequency of occurrence, or types of resources. It explicitly identifies relationships that form chains and composites, which have not been a formal part of past classification efforts. This classification scheme contains a total of 723 CWE's, grouped into 11 main classes. One issue with this classification scheme

is the deep hierarchical structure used: The level of specificity is not the same across all depths. For example, if two CWE's are compared that were taken from two levels beyond different main classes of this classification scheme, one may be very specific to a certain problem and the other may detail a very general set of problems.

CWE-888 is the Software Fault Pattern (SFP) view and is a classification scheme developed as a result of a Department of Defense (DoD) sponsored project through KDM Analytics [37] [38]. This view developed a formal specification of software weaknesses/vulnerabilities that enable automation through focusing on characteristics that are discernible in code, while also ensuring systematic coverage of the "weakness space." The classification schema contains 705 CWE's, grouped into 21 primary and 62 secondary classes. Furthermore, every CWE within this view is classified to exactly one primary and one secondary class, creating a three level hierarchical view. This structure does not have the specificity problem described about CWE-1000. The three levels have well defined levels of specificity with the primary class being the most general, and the third level (individual CWE's) being the most granular. This along with the very intuitive structuring makes this classification a good potential match for this work.

CWE-700 is the Seven Pernicious Kingdoms View which originated from Cigital [39] [40]. This schema offers a simple, effective organization structure for software security coding errors. The creators argure that al other security taxonomies are too complex, due to people on average being good at keeping track of seven (plus or minus two) things. THis provides the motication to reate a taxonomy with only seven primary classes or topics. However, due to this simplified classification structure, only 97 CWE's grouped into seven primary classes are covered in this two level hierarchical structure. This schema is too simplistic and too general for use in this work.

CWE-699 is the Development Concepts view which organizes weaknesses around concepts that are frequently used in software development [41]. Accordingly, this view aligns closely with the perspectives of developers, educators, and assessment vendors. It borrows heavily from the organizational structure used by Seven Pernicious Kingdoms, but it also provides a variety of other categories that are intended to simplify navigation, browsing, and mapping. This classification scheme contains 756 CWE's, organized in a hierarchical structure similar

to CWE-1000. There are six primary classes, which can contain any number of CWE's, subclasses, or nested subclasses. In terms of the viability of using this classification schema for this work, this schema has the same specificity problem as CWE-1000 and will no longer be considered.

The National Vulnerability Database (NVD) integrates the CWE list into the scoring of CVE vulnerabilities by utilizing their own subset of the overall CWE structure [42]. This subset is simply 123 CWE's which the NVD analysts use to assist in scoring CVE's at both a fine a coarse granularity. While this subset may provide good vulnerability coverage, the lack of classes makes classifying an issue into it very difficult, time consuming, and non-trivial. This schema will no longer be considered for this work.

The schema that best fit the needs of this project was the CWE-888 Software Fault Pattern View. CWE-888 includes a three level hierarchical structure of which the first two levels (primary and secondary classes) were used for our classification. This allowed for each issue to have a general (primary) class, and a more specific (secondary) class. Furthermore, the organizational structure of this schema was a good fit for this work as it was developed for enabling automation through focusing on characteristics that are discernible in code. A more complete picture of this schema can be seen in Table A.1. The description of the Software Fault Pattern (SFP) Numbers is out of the scope of this thesis, but can be found from [37].

3.3 Issue Labeling Approach

For each of the datasets, we manually inspected and labeled each issue with its corresponding CWE-888 primary and secondary class. Every issue found to be not security related was assigned a primary and secondary class of "Not Security Related." The fields from the bug tracking systems used for this labeling were the "Title," "Subject," "Description," and if needed the "Recommended Actions" or "Solution." This method allowed for the classification of each issue into two specific levels of detail with the primary class being more general, and the secondary class being more specific.

An interesting problem encountered was that in some cases, even though an issue can

be assigned to a CWE-888 primary and secondary class, it may not necessarily be security related. An example of this could be an issue labeled with the primary class of "Risky Values," and secondary class of "Glitch in Computation." Even though this is a problem and concern, just because a formula is not generating the correct result does not mean that this issue is a security concern. Additionally, we did not have the necessary information to determine if the security related issues could be exploited or what the overall security related impact on the system would be. Here, similarly to labeling done by static code analysis tools, we took a conservative approach and treated as security related every issue to which a CWE class could be assigned. Several Examples of this classification follow:

The description of an issue read ". . . Line 277: Null pointer dereference of 'getServiceStatusInfo(...)' where null is returned from a method," then this issue was labeled with the primary cluster of "Memory Access" and the secondary cluster of "Faulty Pointer Use."

An issue description read ". . . Table 1-11 lists XYZ as a unidirectional interfaces, but Figure 1-4 shows this connection as bidirectional," then this issue was labeled as "Not security Related."

As a final example, an issue description read ". . . The stream is opened on line 603 of file1. If an exception were to occur at any oint before line 613 where it is closed, then the 'try' would exit and the stream would not be closed," then the issue was labeled with the primary class of "Resource Management" and the secondary class of "Failure to Release Resource."

Upon completion of the labeling, each dataset was analyzed. The results and conclusions are detailed in the following sections.

3.4 Ground Mission IV&V Issues

Of the 1779 issues in this dataset, 350 (20%) were marked as potentially security related by the IV&V analysts. After labeling, it was determined that 133 of the 350 were truly security related (38% of the original 350). This reduction in security related issues is due to a large concentration of testing issues. A testing issue is an issue detailing a problem with a testing system instead of a problem with the system being tested. No CWE's exist that

cover such a case and testing issues are not dealing with the actual system under concern, therefore we labeled these issues as "Not Security Related."

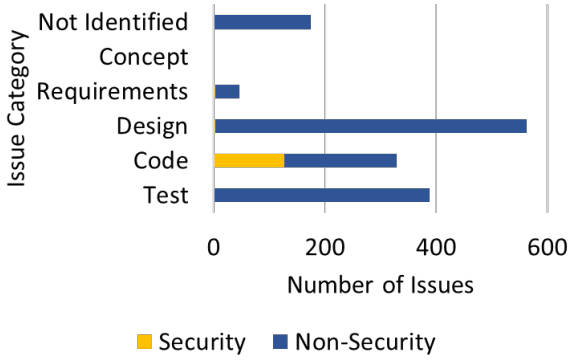

Figure 3.1: Ground Mission IV&V Issues - Issue Category Distribution

Figure 3.1 shows the distribution of security and non-security issues across the different issue categories. The issue category defines what aspect of the project the issues falls under. As shown, the Design category contained the highest number of issues. However, this category consists of only 2.3% (three of 133) of all security issues. The code category houses the vast majority of security issues, containing 95.5% (127 of 133) of all security issues.

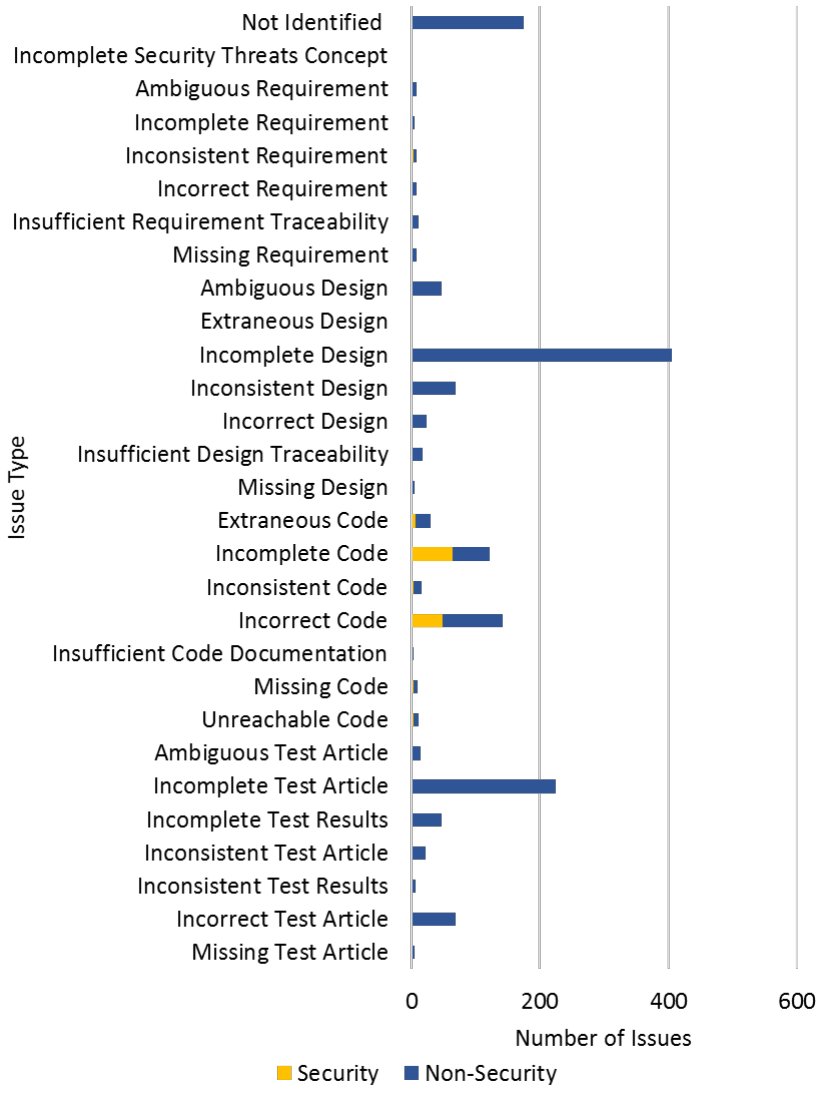

Figure 3.2: Ground Mission IV&V Issues - Issue Type Distribution

Figure 3.2 shows the distribution of security and non-security issues across the different issues types, which are most detailed than the issue category. Four dominating issue types are seen to be "Incomplete Design," "Incomplete Code," "Incorrect Code," and "Incomplete Test Article." Similar to the issue category, none of the "Incomplete Test Article" issues are security related and only one "Incomplete Design" issue is security related. The code related issues types of "Incomplete Code" and "Incorrect Code" contain 84% (112 of 133) of security related issue. Furthermore, 43% of the issues in "Incomplete Code" and "Incorrect Code" are security related.

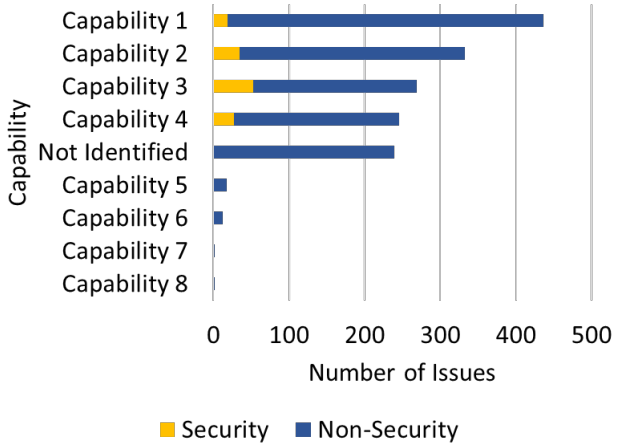

Figure 3.3: Ground Mission IV&V Issues - Capability Distribution

Figure 3.3 shows the distribution of the issue capabilities ordered from the capability that contains the highest total number of issues to the capability that contains the least. Capability 1, 2, 3, and 4 hold all of the security issues. Capability 3 has the most security issues containing 40% (53) of the 133 security issues, followed by Capability 2 with 26% (34) of the 133 security issues. Although only half of the Capabilities house all of the security related issues, they are also represent 82% of all Capabilities used. Each capability that contains security issues have a similar ratio of security to non-security issues.

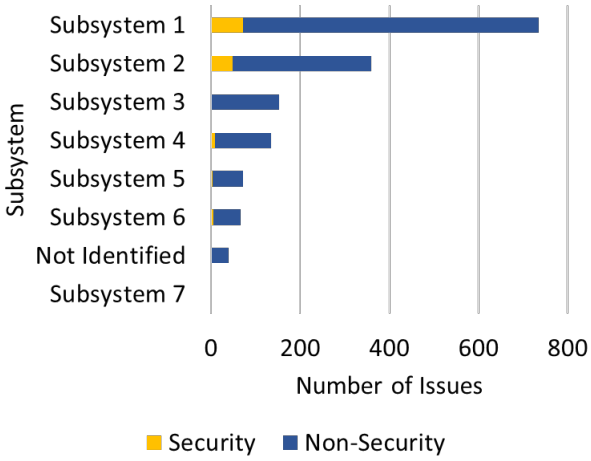

Figure 3.4: Ground Mission IV&V Issues - Subsystem Distribution

Figure 3.4 shows the subsystem distribution ordered from the subsystem that contains

the highest total number of issues to the subsystem that contains the least. Subsystem 1 and 2 contribute 70% of all issues, which is consistent with the Pareto Principle. The Pareto Principle states that for many events, roughly 80% of the effects come from 20% of the causes. In this case, 70% of all issues, and 86% of all security issues come from two of the seven subsystems which is consistent with the results shown in [3], [43], [1], and [9].

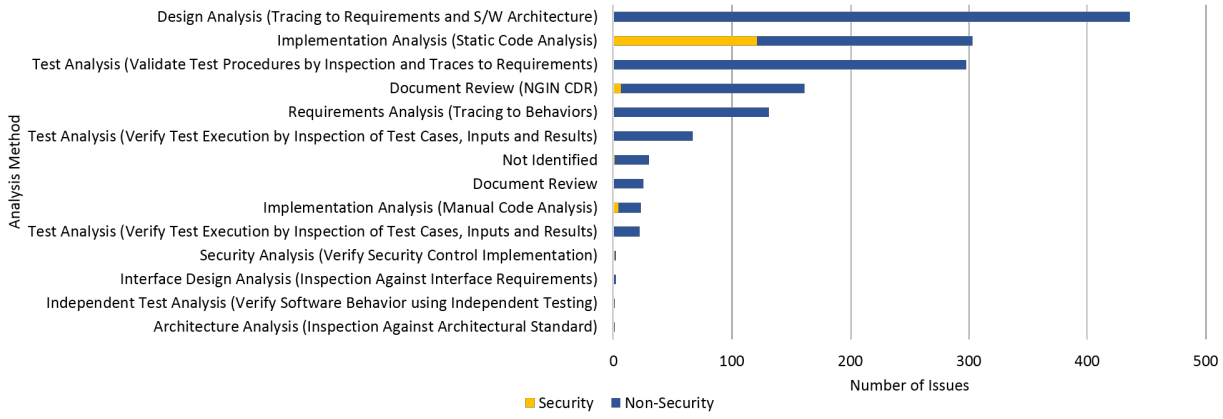

Figure 3.5: Ground Mission IV&V Issues - Analysis Method Distribution

Figure 3.5 shows the distribution of issues with respect to the analysis method used to detect the issues. The largest proportion of issues (30% of all issues) was found from "Design Analysis," however this method did not uncover any security issues. The vast majority of security issues were discovered using "Implementation Analysis (Static Code Analysis)." This method was used to discover 91% of all security related issues. Interestingly, the analysis method of "Security Analysis (Verify Security Control Implementation)" turned up almost no issues. This is possibly due to the difficulty of determining the potential security problems in the first place. If a developer is aware of a problem then they are easily able to fix it, however when a problem is not known, it cannot be fixed. The security analysis method highlights this difference as the problems the analysts know to look for have already been fixed. Furthermore, the largest portion of security issues were found via Static Code Analysis.

The analysis method used can greatly effect the results of this analysis. Often times, the results are specific to a specific tool or method, and when a different tool or method is used

different results are presented. The amount of time and energy invested in each method could also influence its effectiveness. The only information pertaining to the amount of effort expended on each method known is that significant amount of static code analysis was preformed, and therefore a complete picture of the effectiveness of each analysis cannot be drawn from this information.

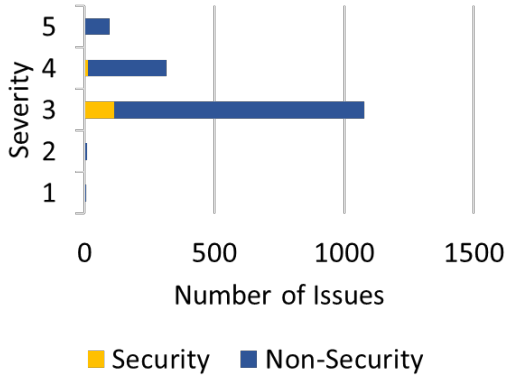

Figure 3.6: Ground Mission IV&V Issues - Severity Distribution

NASA severity ratings range from 1 to 5, with 1 being the most severe. The majority of all issues (72%) are of severity 3 as shown in Figure 3.6. This trend remains the same for security issues as well, with 86% of all security related issues being of severity 3.

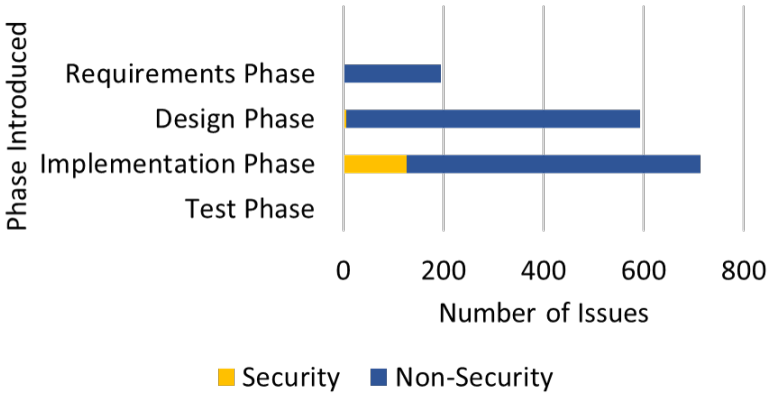

Figure 3.7: Ground Mission IV&V Issues - Phase Introduced Distribution

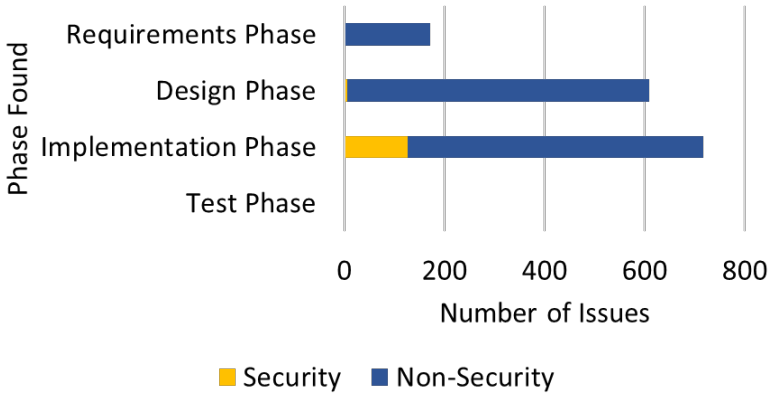

Figure 3.8: Ground Mission IV&V Issues - Phase found Distribution

Figure 3.7 and 3.8 detail the phase in which each issue was introduced and found in the project. The majority of security issues (91%) were introduced in the implementation phase, which again shows how hard implementing security code is compared to determining the requirements and design from a security standpoint. Furthermore, the phase in which issues were found closely followed the phase in which they were introduced.

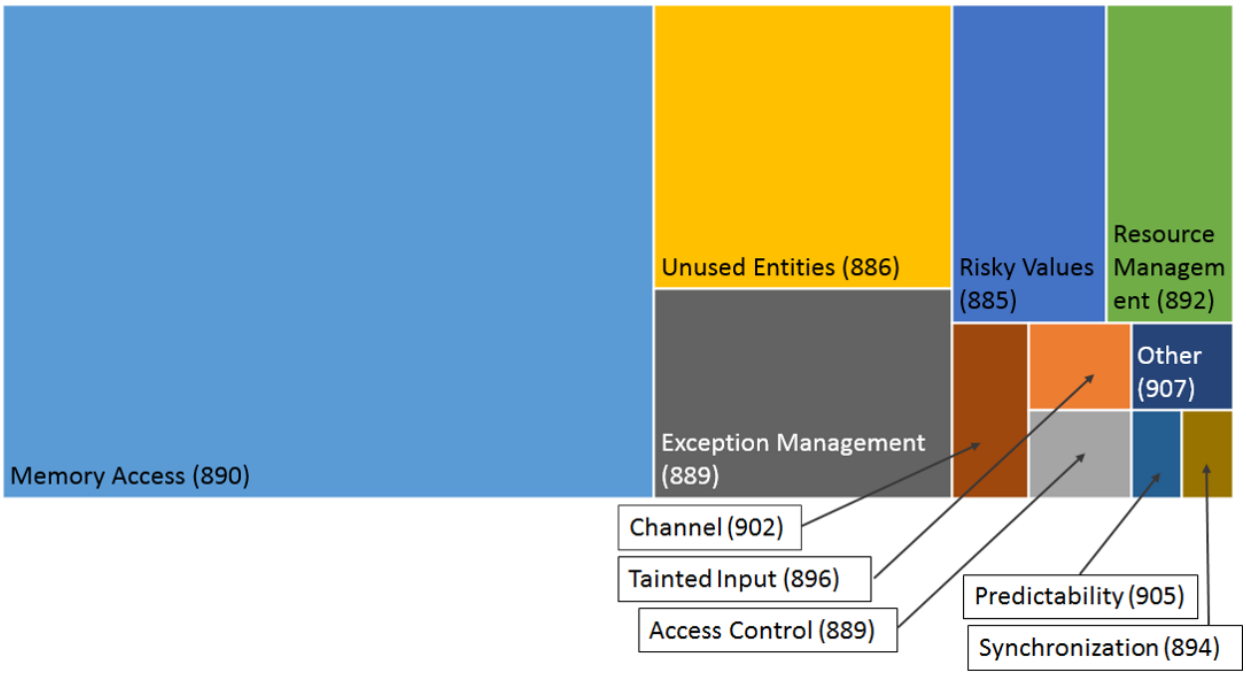

Figure 3.9: Ground Mission IV&V Issues - Distribution of issues across CWE-888 Primary Classes

Figure 3.9 shows the distribution of security issues across the primary CWE-888 classes, however only 11 out of the 21 primary classes were observed. The class of "Memory Access" consisted of 53% of all security issues. Furthermore, the primary classes of "Memory Access," "Unused entities," "Exception Management," "Risky Values," and "Resource Management" contain 92% of all security issues.

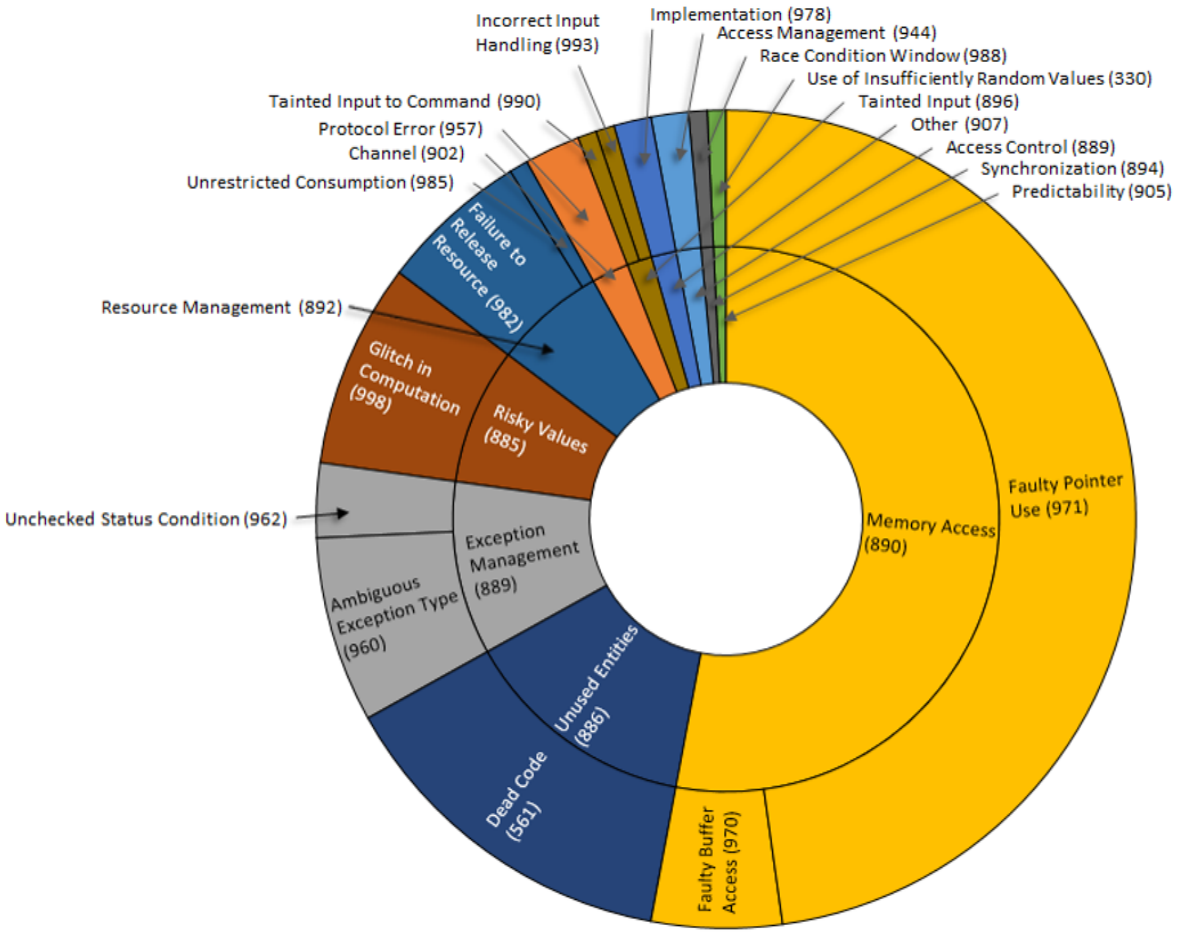

Figure 3.10: Ground Mission IV&V Issues - Primary and Secondary CWE-888 Class Distributions

The secondary CWE-888 classes provide more detail than simply the primary classes. Figure 3.10 shows the overall distribution of security issues in both the primary and secondary class, which better represents the types of security issues. This figure shows that of the 70 issues in the "Memory Access" primary class, 63 are of the secondary class "Faulty Pointer Use." The remaining dominating class of "Unused Entities," "Exception Management," and

"Risky Values" were comprised mainly of the secondary classes "Dead Code," "Ambiguous Exception Type," and "Glitch in Computation" respectively.

3.5 Flight Mission IV&V Issues

After the removal of "Withdrawn" or "Not an Issue" issues, 383 issues remained. After labeling, 157 issues were marked as security related (41% of all issues). The following figures show the results of the analysis.

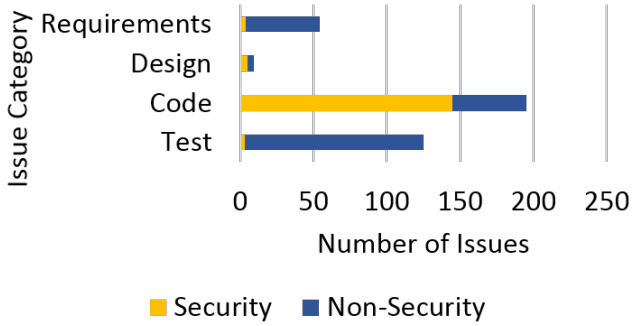

Figure 3.11: Flight Mission IV&V Issues - Issue Category Distribution

As shown in Figure 3.11, most of the security related issuse were associated with the "Code" category, which contributed 92% of all security related issues. This project did not have the testing issue problem as described in the previous section, which is a result of no issue in this dataset containing any specific security related tags but all issues being manually classified into the CWE-888 classification scheme. This distribution of security related issues aligns with the conclusions presented in the previous section, with majority of security issues related to the implementation, rather than early life cycle phases.

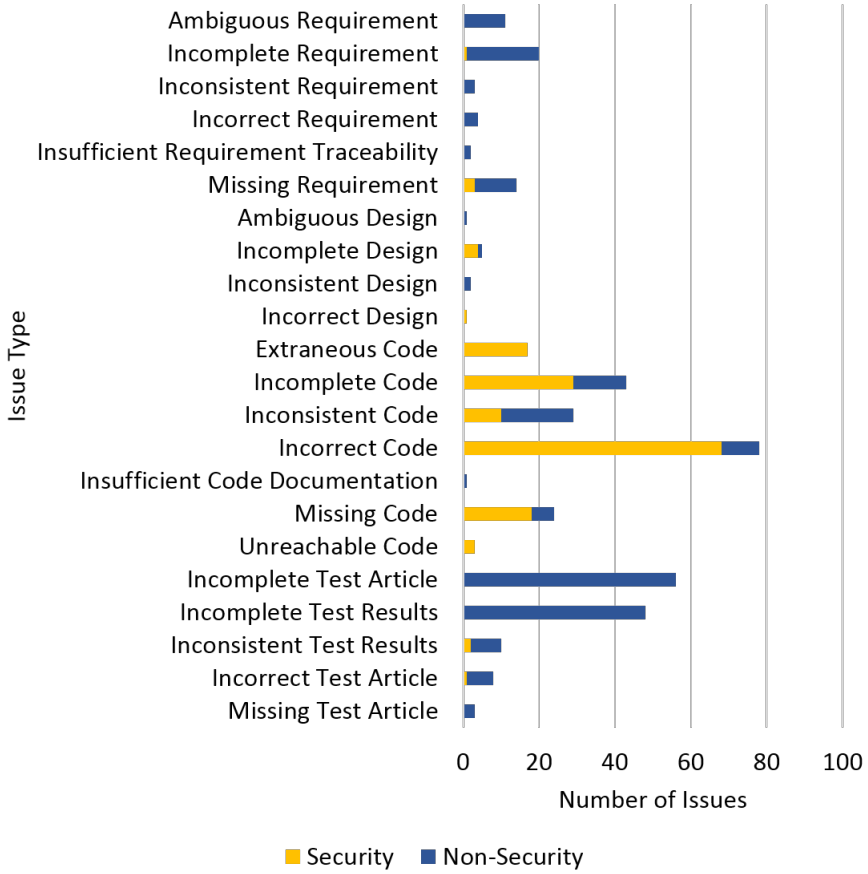

Figure 3.12: Flight Mission IV&V Issues - Issue Type Distribution

Figure 3.12 shows four issue types dominating the majority of the security issues: "Incorrect Code," "Incomplete Code," "Missing Code," and "Extraneous Code." It is not surprising that these dominating issues types are all code related having in mind that code issue category had most of the security issues.

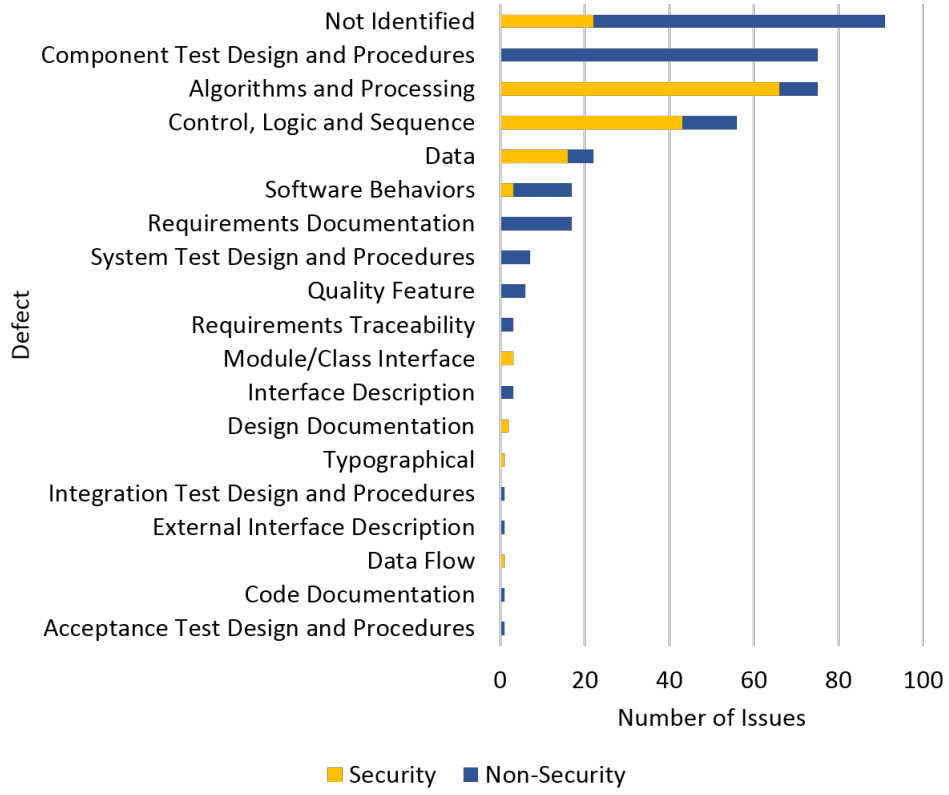

Figure 3.13: Flight Mission IV&V Issues - Defect Distribution

Figure 3.13 provides a breakdown by defect categories. The three dominating defect categories are "Algorithms and Processing," "Control, Logic and Sequence," and "Data."

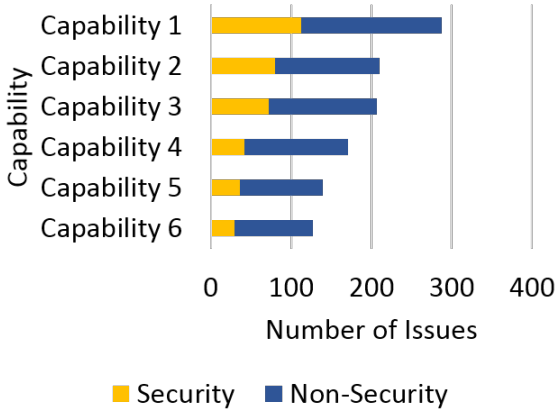

Figure 3.14: Flight Mission IV&V Issues - Capability Distribution

Figure 3.14 is not very informative. Each capability contains approximately the same

proportion of security issue with respect to its size (i.e. each capability is approximately 30% security and 70% non-security). Even though security issues are more concentrated in certain development phases or types (i.e. the implementation or coding), the functionality of the code being created has little to no impact on the number of security issues associated with it.

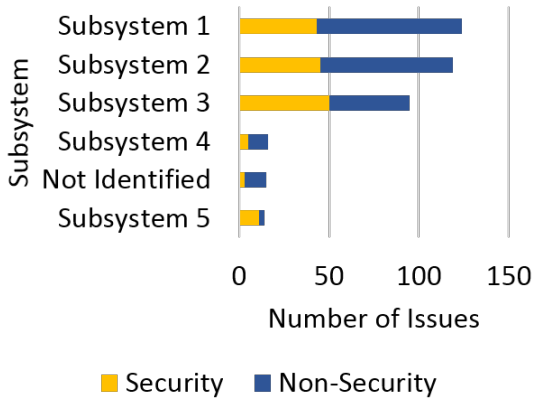

Figure 3.15: Flight Mission IV&V Issues - Subsystem Distribution

Figure 3.15 shows that 88% of all security issues and 88% of all issues fall into three subsystems. Furthermore, according to this figure, Subsystem 3 is the most security issue prone with 53% of the issues related to this subsystem being security related.

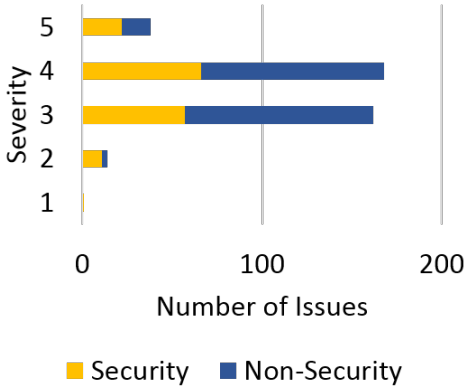

Figure 3.16: Flight Mission IV&V Issues - Severity Distribution

As shown in Figure 3.16, severity levels 3 and 4 contain 79% of all security issues and 86% of all issues.

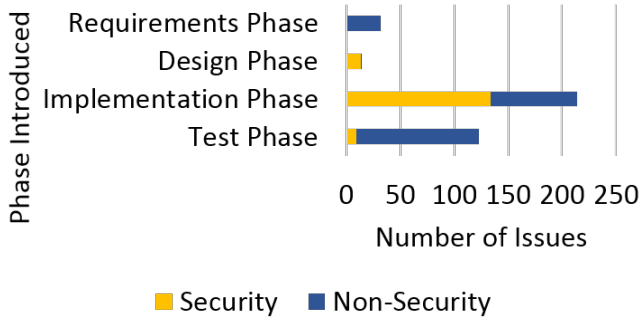

Figure 3.17: Flight Mission IV&V Issues - Phase Introduced Distribution

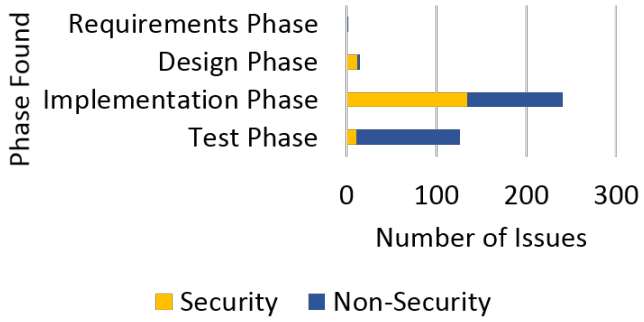

Figure 3.18: Flight Mission IV&V Issues - Phase Found Distribution

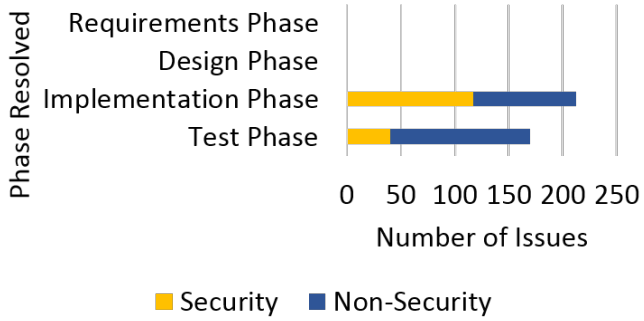

Figure 3.19: Flight Mission IV&V Issues - Phase Resolved Distribution

Figures 3.17 and 3.18 show results consistent with the Ground Mission IV&V issues, where the majority of security issues were introduced (85%) and found (85%) in the implementation phase. The phase in which an issue was found closely follows the phase in which

the issue was introduced. Unlike the Ground Mission IV&V Issues dataset, the Flight Mission IV&V Issues dataset included information on the phase in which each issue was resolved, detailed in Figure 3.19. 75% of security related issues were resolved in the implementation phase, and the remaining 25% were resolved in the testing phase. Interestingly enough, no security issues were resolved in the design phase, even though some were introduced and found in this phase.

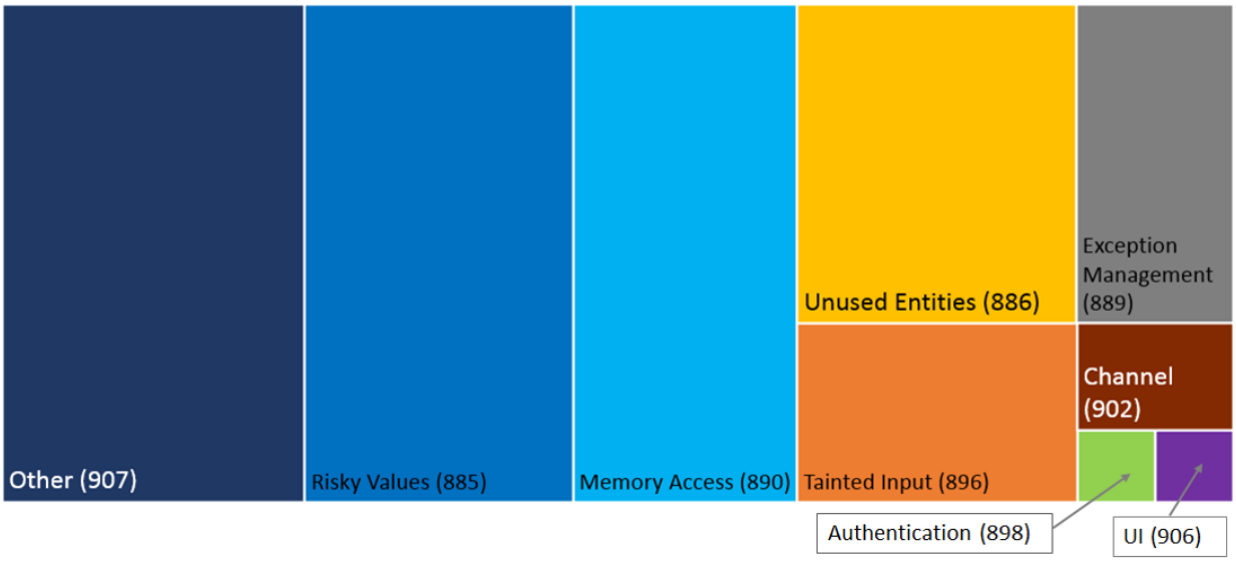

Figure 3.20: Flight Mission IV&V Issues - Distribution of issues across CWE-888 Primary Classes

The diagram shown in Figure 3.20 shows the distribution of security issues when labeled with their corresponding CWE-888 primary class. Similarly as the Ground Mission IV&V Issues dataset, only 9 of the 21 primary classes were observed with the dominating classes of "Other," "Risky Values," "Memory Access," and "Unused Entities." .

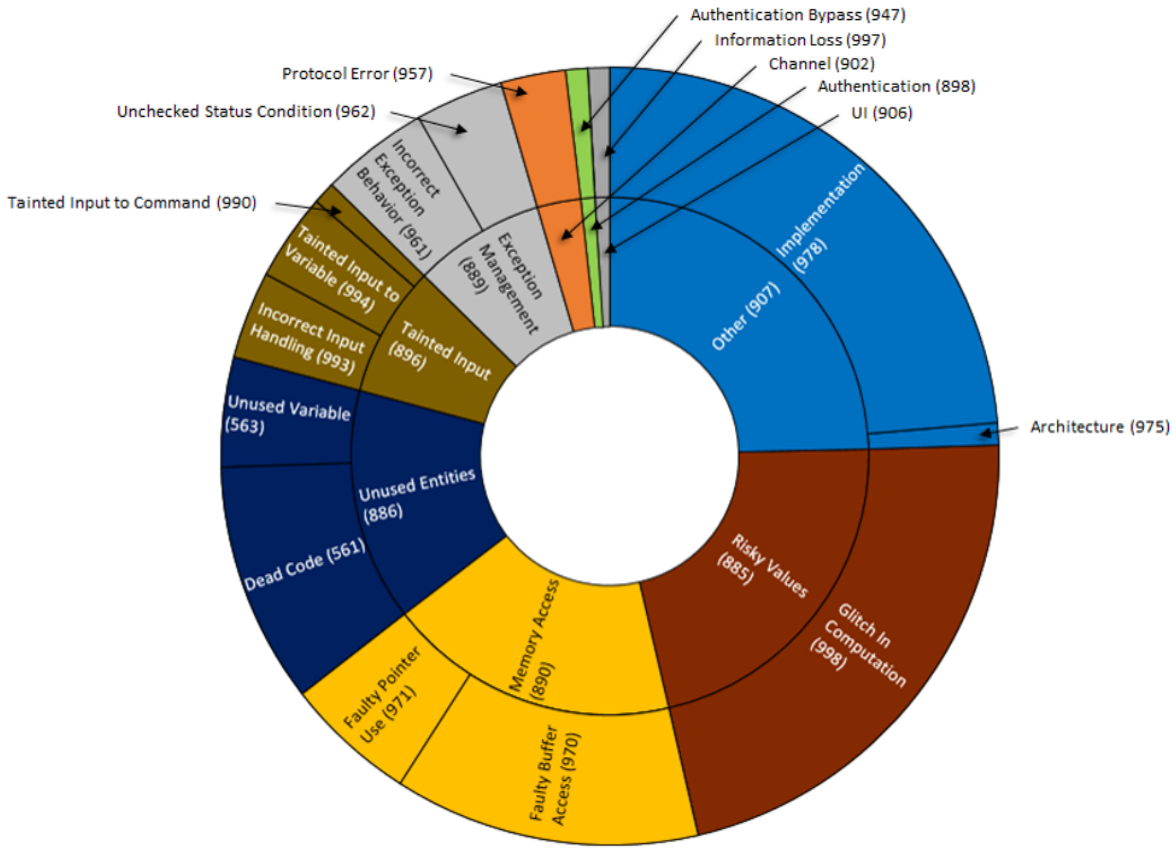

Figure 3.21: Flight Mission IV&V Issues - Primary and Secondary CWE-888 Class Distributions

Figure 3.21 shows the distribution of security issues across the primary and secondary classes. The secondary classes provide more detail on the types of security issues seen throughout this dataset than the primary classes, giving a more specific picture of the types of security issues observed. Of the 59 "Other" issues, 55 were of the secondary class "Implementation." The primary class of "Risky Values" contained 30 issues, all of which with the secondary class of "Glitch in Computation." Of the 20 "Memory Access" issues, 14 were of the secondary class "Faulty Buffer Access." 'The primary class of "Unused Entities" contained 23 issues, 18 were of the secondary class "Dead Code."

3.6 Flight Mission Developer Issues

After the removal of all issues that were not "Closed" and a "Defect," 569 issues remained. Labeling resulted in 374 of these issues marked as security related (66% of all issues), which is significantly higher than the proportion of security related issue in the other datasets. The following figures show the results of the analysis.

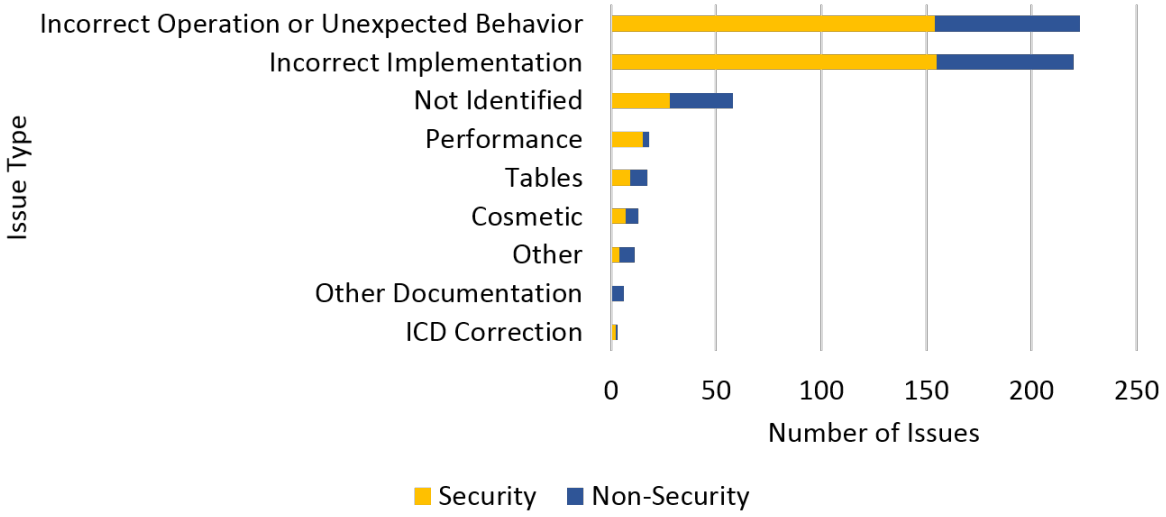

Figure 3.22: Flight Mission Developer Issues - Issue Type Distribution

Figure 3.22 shows the distribution of issue types. Two dominating categories are "Incorrect Implementation" and "Incorrect Operation or Unexpected Behavior." The dominating category "Incorrect Implementation" is consistent in what has been seen in the previously analyzed datasets.

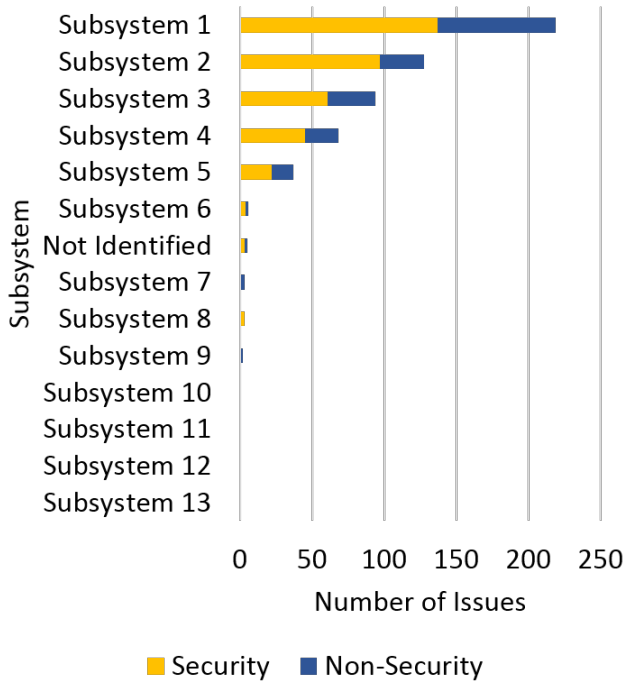

Figure 3.23: Flight Mission Developer Issues - Subsystem Distribution

Figure 3.23 presents the distribution of security and non-security issues across the subsystems used in this dataset. The characteristics are very similar to the previous datasets, with 88% of all security issues found in only four subsystems, which together houses 89% of all issues.

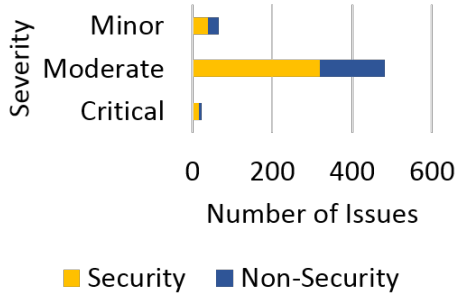

Figure 3.24: Flight Mission Developer Issues - Severity Distribution

While the severity ratings used by the IV&V analysts ranged from 1 to 5, the ratings found in this dataset are Minor, Moderate, and Critical. In a fashion similar to previously

observed the moderate category dominated, containing 86% of the security issues, and 85% of all issues. Only 4% of all issues, and 4% of security issues were determined to be critical.

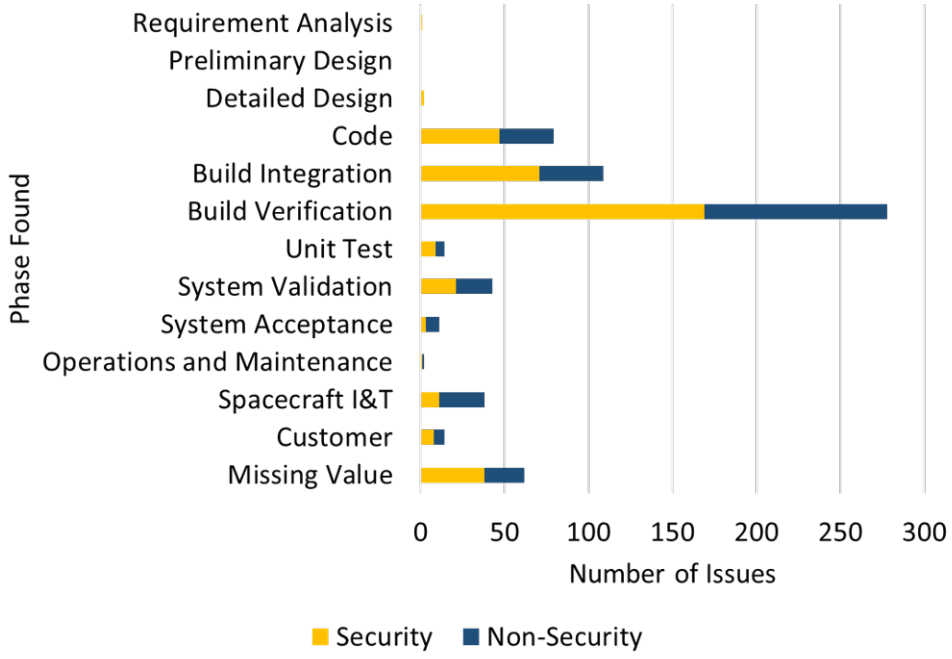

Figure 3.25: Flight Mission Developer Issues - Phase Found Distribution

This dataset contained information about the phase in which each issue was found, yet no information on when they were introduced or resolved. As shown in Figure 3.25, the major phases where issues were found were "Build Verification," "Build Integration," and "Code Implementation." This is similar to what has been previously observed in the previous datasets.

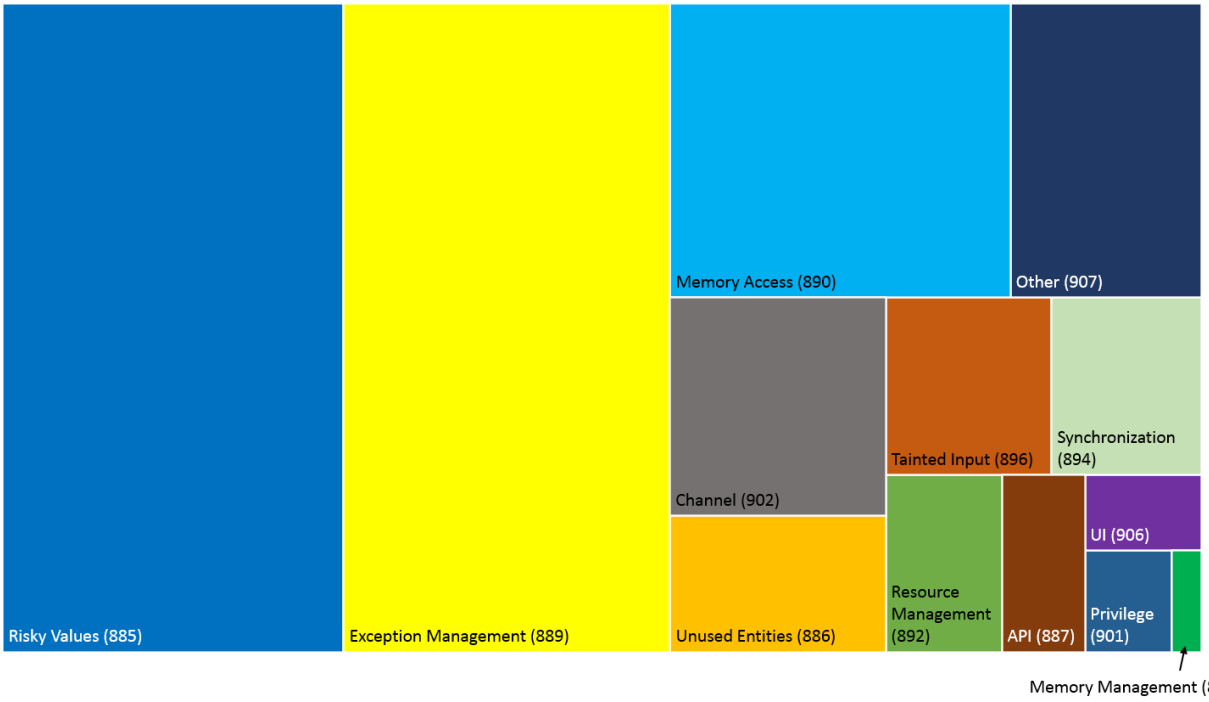

Figure 3.26: Flight Mission Developer Issues - Distribution of Issues Across CWE-888 Primary Classes

The diagram shown in Figure 3.26 shows the distribution of security issues when placed into corresponding CWE-888 class. While only 13 of 21 primary class were observed, "Risky Values," "Exception Management," and "Memory Access" were responsible for 50% of all security issues.

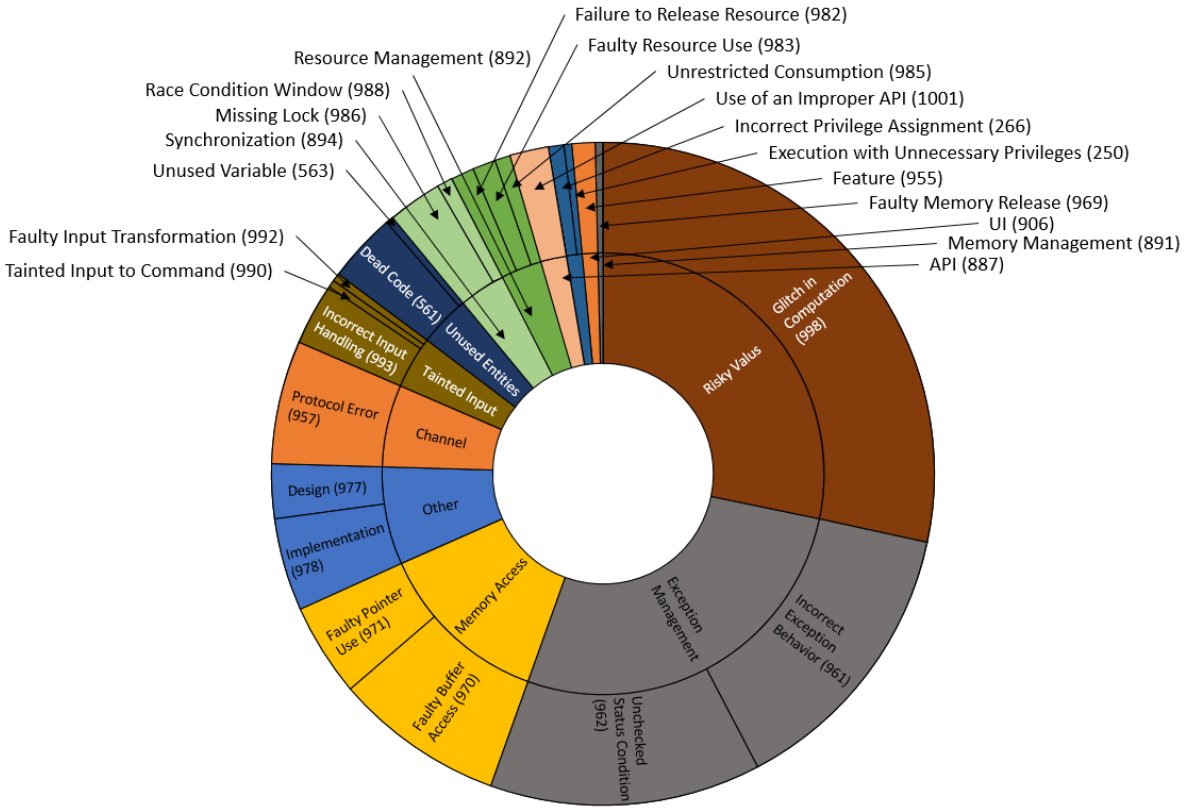

Figure 3.27: Flight Mission Developer Issues - Primary and Secondary CWE-888 Class Distributions

Figure 3.27 show the distribution of security issues across the primary and secondary classes. The secondary classes provide more detail on the types of security issues seen throughout this dataset then the primary classes, giving a more specific picture of the types of security issues observed. The previous datasets saw mostly a single secondary class dominating within each primary class, which is not always the case with this dataset. The primary class with the most issues is "Risky Values" and all issues of this primary class are of the secondary class of "Glitch in Computation." However, the next largest primary class is "Exception Management" with 90 issues: of these 90 issues, 48 were of the secondary class "Incorrect Exception Behavior," and the remaining 42 were of the secondary class "Unchecked Status Condition." A similar distribution occurs within the primary class of "Memory Access" where of the 34 issues, 22 are of the secondary class "Faulty Buffer Access," and the remaining 12 are of the secondary class "Faulty Pointer Use."

3.7 Comparison of the Results Across Three Datasets

This section presents a comparison of the results across all datasets. The CWE-888 primary and secondary classes are used to compare the security distribution across projects, as well as the issues most often plaguing NASA systems. Furthermore, the main findings across these datasets are presented.

Table 3.1: Comparison of Primary Class Distributions Across All Projects

Primary CWE-888 Class	**Ground Mission IV&V Issues**	**Flight Mission IV&V Issues**	**Flight Mission Developer Issues**
API (887)			1.9%
Channel (902)		2.7%	6.0%
Exception Management (889)	**10.8%**	**8.2%**	**27.2%**
Memory Access (890)	**54.6%**	**18.3%**	**12.8%**
Memory Management (891)			0.4%
Other (907)	1.5%	**24.5%**	7.1%
Predictability (905)	0.8%		
Privilege (901)			1.2%
Resource Management (892)	6.9%		3.0%
Risky Values (885)	**8.5%**	**21.8%**	**28.3%**
Synchronization (894)	0.8%		3.4%
Tainted Input (896)	1.5%	**8.2%**	3.8%
UI (906)		0.9%	1.1%
Unused Entities (886)	**14.6%**	**14.5%**	3.8%

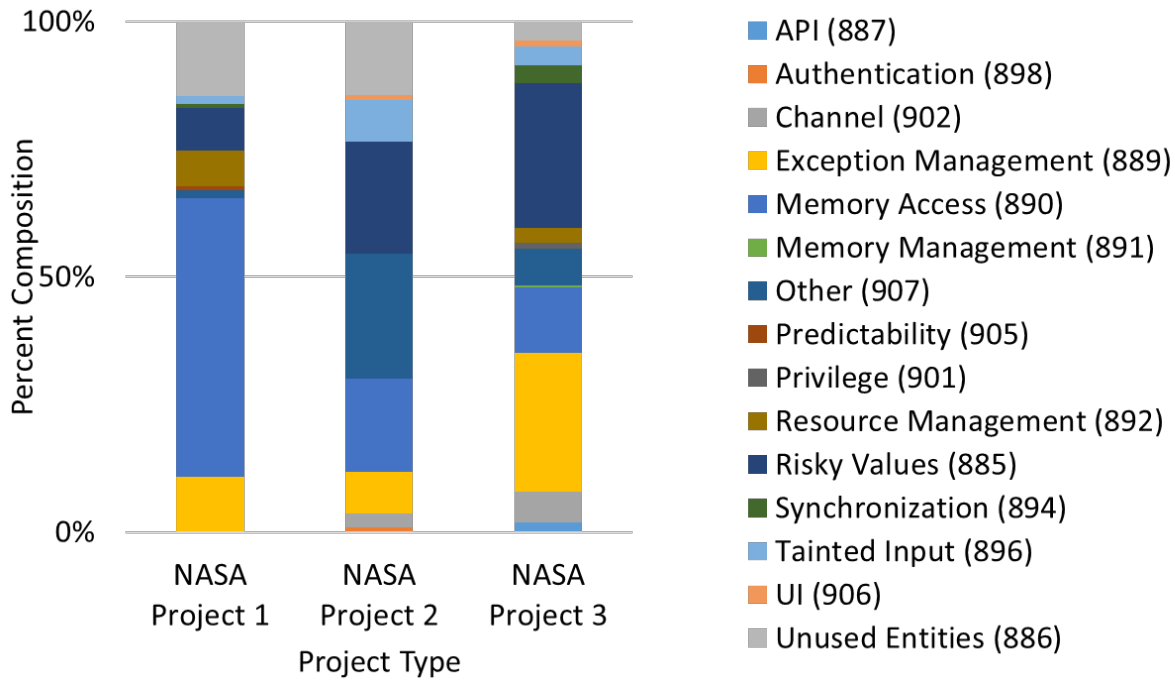

Figure 3.28: CWE-888 Primary Class Distribution Graphical Comparison for all Projects

Table 3.1 and Figure 3.28 detail the percent in which each CWE-888 primary classes contributes to each dataset. The primary classes of "Exception Management," "Memory Access," "Other," "Risky Values," and "Unused Entities" account for the most significant portion of security issues across all projects. Interestingly, primary classes which do not occur in all datasets, tend to make up for only a small proportion (7% or less) of the datasets they do appear in.

Table 3.2: Comparison of Dominating Secondary CWE-888 Class Distributions Across All Projects

Primary CWE-888 Class **Secondary CWE-888 Class**	**Ground Mission IV&V Issues**	**Flight Mission IV&V Issues**	**Flight Mission Developers Issues**
Exception Management (889)			
Ambiguous Exception Type (960)	7.7%		
Incorrect Exception Behavior (961)		4.5%	**14.0%**
Unchecked Status Condition (962)	3.1%	3.6%	**13.2%**
Memory Access (890)			
Faulty Buffer Access (970)	4.6%	**12.7%**	8.3%
Faulty Pointer Use (971)	**50.0%**	5.5%	4.5%
Other (907)			
Architecture (975)		0.9%	
Design (977)			2.6%
Implementation (978)	1.5%	**23.6%**	4.5%
Risky Values (885)			
Glitch in Computation (998)	8.5%	**21.8%**	**28.3%**
Unused Entities (886)			
Dead Code (561)	**14.6%**	**10.0%**	3.4%
Unused Variable (563)		4.5%	0.4%

Table 3.2 shows the dominating CWE-888 primary classes along with their corresponding secondary class across all projects. The secondary classes most often occurring in "Exception Management" issues were "Ambiguous Exception Type," "Incorrect Exception Behavior," and "Unchecked Status Condition." These secondary class names are specific, self explanatory, and will not be discussed further.

"Memory Access" was another dominating class, consisting of "Faulty Buffer Access" and "Faulty Pointer Use." These categories include common programming errors such as null pointer dereferences and buffer overflows. As shown by Younan [18], buffer overflows continue to be one of the most common vulnerabilities in software systems.

The next dominating primary class is "Other" with the most commonly seen secondary classes of "Design" and "Implementation." "Design" consists of weaknesses dealing with insufficient control flow management or reliance on data/memory layout. "Implementation" is based around weaknesses such as coding standards violation or containment errors. These secondary classes were assigned to issues which had security related problems related strictly

to their design or implementation and could not be placed into any other class.

Another dominating primary class is "Risky Values." This primary class consists of the secondary class "Glitch in Computation" which deals with calculation errors. These involve everything from a divide by zero error to a function call with an incorrect order of arguments. The Flight Mission Developer Issues dataset had the highest percentage of these errors. This is potentially be because this was the only bug tracking system from developers. The developers are more likely to focus on things such as the incorrect generation of results, and therefore fix them before they reach the IV&V analysts (in this case Flight Mission IV&V Issues).

The last consistently dominating class is "Unused Entities," consisting of the "Dead Code" and "Unused Variable" secondary classes. These issues were abundant across all project types, but were found much more often in the IV&V datasets.

Table 3.3: Main Findings Across all Datasets

	Ground Mission IV&V Issues	Flight Mission IV&V Issues	Flight Mission Developers Issues
% Security Issues	9%	41%	66%
Security Issues Category	95% Code	92% Code	Data not available
Severity of Security Issues	Level 3 dominated (86%)	Levels 3 and 4 dominated (together 78%). 7% were level 2	Moderate dominated (84%)
Phase Introduced	95% in the Implementation Phase	85% in the Implementation Phase	Data not available
Phase Found	Followed closely the phase introduced distribution	Followed closely the phase introduced distribution	Most found during Code Implementation, Build Integration, and Build Verification
Subsystem	86% found in two subsystems (70% of all issues)	88% in three subsystems (88% of all issues)	88% in four subsystems (90% of all issues)
Five (out of 21) most frequent Primary Classes	Exception Management 10.8% Memory Access 54.6% Other 1.5% Risky Values 8.5% Unused Entities 14.6% Total 90.0%	Exception Management 8.2% Memory Access 18.2% Other 24.5% Risky Values 21.8% Unused Entities 14.5% Total 87.3%	Exception Management 27.2% Memory Access 12.8% Other 7.1% Risky Values 28.3% Unused Entities 3.8% Total 79.2%

Table 3.3 shows the main findings for each dataset. The percentage of security issues for each dataset was covered a wide range (from 9% to 66%). One possibility for this is that the Ground Mission IV&V Issues dataset is a sample from a not yet complete project, as the

testing phase has not yet begun. However, in Flight Mission IV&V Issues the majority of all security issues were introduced and found in the implementation phase, which indicates that the Ground Mission IV&V Issues dataset simply had less security related issues.

Not surprisingly, the Ground Mission IV&V Issues dataset as well as the Flight Mission IV&V Issues dataset had the vast majority of security issues belonging to the "Code" category (95% and 92% respectively. The Flight Mission Developers Issues dataset however did not contain this information. This is to be expected as implementing secure code (instead of developing the requirements or design) proves to be the most difficult aspect of software security.

Ideally, security issues should be of higher severity than non-security. However, across all datasets, the majority of all issues along with the majority of security issues were of a severity level relating to moderate. This is most likely due to the severity being assigned to an issue before the security implication of that issue are known or realized.

Furthermore, security issues should be fixed in a timely manor to not only minimize the time that vulnerabilities exist in the system, but to also prevent its propagation. While the data pertaining to the project phase in which issues were found was not given in the Flight Mission Developers Issues dataset, in both the Ground Mission IV&V Issues dataset and the Flight Mission IV&V Issues dataset the phase in which an issue was found closely follows the phase in which it was introduced. This correlates to security issues being found during the same phase in which they are introduced, and therefore being identified in a timely fashion.

The subsystem is a grouping of software components, juxtaposed to create a modular piece of code to accomplish a specific task. As these subsystems can be reused, it is important to determine the risk of introducing security related issues into the system by incorporating them. This is hard to determine using only the bug tracking system as the amount each subsystem is used is unknown. However, each dataset shows two to four subsystems which contain the majority of security issues, and the majority of all issues. Assuming each subsystem is used in proportion to the total number of issues found for it, then each subsystem introduce very similar risk into the system through their implementation.

The final row in Table 3.3 details the previously discussed dominating primary CWE-888 class, along with the percent make up of each dataset. For a more detailed description and

analysis of the distribution of these issues across each dataset, see Table 3.2 and the following discussion.

3.8 Threats to Validity

Many issues arose during the analysis of each project. One problem with each data set is that it originates from human analysis. This creates problems because the number of security related issues depends on the quality of the software artifacts, the validation and verification (V&V) methods used, and the amount of effort expended in the issue creation, classification, and analysis. Therefore, vulnerability profiles built from projects bug tracking systems depend on the quality and completeness of information provided in the bug tracking system.

Furthermore, some instances arose when a bug report could be correctly classified into multiple CWE-888 classes. This has no clear solution and was solved by selecting the most relevant of the possible classes. Although this can change the vulnerability type distribution, it most likely has little to no effect as the issue is still accounted for, and the number of issues fitting into multiple classes was small.

As described in Section 3.4, testing issues marked as security related by IV&V analysts were not included in out analysis. A testing issue details a problem with a testing system, instead of a problem with the system being testing. No CWE's exist that cover such a case and therefore could not be included in the analysis.

Each dataset contained a small amount (15% or less) if issues that did not contain sufficient information for classification. To classify these issues an analyst or developer would need to look further into the issue and provide more details. Issues which did not contain the information necessary for classification to CWE classes were not included in the analysis.

3.9 Conclusion

We conducted the empirical analysis of the NASA datasets to determine the vulnerability distributions and trends. We summarize the research questions 1 and 1a here as follows:

1. What are the dominate types of vulnerabilities in NASA ground and flight software systems?

    a) Are they consistent across projects and project types?

The dominate types of vulnerabilities in NASA ground and flight software systems are Exception Management, Memory Access, Other, Risky Values, and Unused Entities accounting for 79% to 90% of all security issues, depending on the dataset. Looking further into the secondary classes of these dominating types provided more detail of the exact vulnerability types as follows: the most commonly observed vulnerabilities among Exception Management issues were Ambiguous Exception Type and Incorrect Exception Behavior. The most commonly observed vulnerabilities among Memory Access issues were Faulty Buffer Access and Faulty Pointer Use. The most commonly observed vulnerability among Other issues was Implementation. The most commonly observed vulnerability among Risky Values issues was Glitch in Computation. The most commonly observed vulnerability among Unused Entities issues was Dead Code.

While the dominating vulnerability types differ in which contains the highest percentage of security issues in each dataset, the same five vulnerability types consistently dominate across ground and flight missions as well as across the IV&V and Developer Datasets.

Chapter 4

Automatic Issue Classification

The vulnerability profile described in Chapter 3 provides valuable information to the analysts and developers that aim to increase the security of the software system under concern. However, developing such vulnerability profiles required manual classification of each issue, which is time consuming and costly. This section addresses this concern by using machine learning techniques to automatically classify bug report. Research questions 2, 2a, 2b, 3, 3a, 4, and 5 will be addressed.

2. Can supervised machine learning algorithms be used to classify software issues as security related or not?

    a) Do some learners preform consistently better than others?

    b) How much data must be set aside for training in order to produce accurate classification results?

3. Can supervised machine learning algorithms be used to classify security issues to specific security classes?

    a) Are some classes harder to predict than others?

4. Can unsupervised machine learning algorithms be used to classify software issues as security related or not?

5. How does the performance of supervised and unsupervised machine learning algorithms compare when classifying software bug reports?

4.1 Datasets, Data Extraction, and Preprocessing

The datasets used throughout this chapter are the same datasets as described in Section 3.1. Although these datasets contained many fields as described in Tables B.1 and B.2, only the Title, Subject, and Description of each issue were used for the automated classification. These fields were selected because every issue includes them, and they do not depend on data that is only available after extensive human analysis such as the "Recommended Solution." Furthermore, the CWE-888 class of each issue was kept as the class label.

Specifically, the Title, Subject, and Description of each issue were extracted, and then concatenated into a single string. The preprocessing steps of removing all non-alphanumeric characters using a regular expression in python, converting all characters to lowercase with python, remove stop words using python's Natural Language Toolkit (NLTK) English stop word list [44], and then stem each word with python's Lovins stemming algorithm implementation [45].

After all of the preprocessing steps were completed, we were left with one string for each issue in the dataset. The features to be used for the machine learning were then extracted from these strings as described in the next section.

4.2 Feature Vectors

Three tyes of feature vectors were used throughout this project: Binary Bag-of-Words Frequency (BF), Term Frequency (TF), and Term Frequency-Inverse Document Frequency (TF-IDF). Traditional terminology when discussing these methods include terms, documents, and corpus. This work has three corpora, one for each dataset and will be denoted in the same manor as the datasets they originated from: Ground Mission IV&V Issues, Flight Mission IV&V Issues, and Flight Mission Developers Issues. A "term" is a word within a document, or in this case a word in the string representation of an issue. A document is a

collection of terms, or in this case a string representing an issue. A corpora is a collection of documents, or in this case the collection of strings representing an issue from a specific bug tracking system. From here on, term, document, and corpus will refer to the aforementioned definitions.

All of the aforementioned feature extraction methods produce a vector of numeric values for each document. Each method also does this in a similar way, by each location in the vector representing a term, and the numeric value at the location representing the occurrence of that term in the document. The words which each location in these feature vectors represent is referred to as the vocabulary. This is an important aspect of these feature extraction methods. Selecting a large vocabulary would improve the coverage, and therefore the amount of terms extracted from each document analyzed; however, this leads to a very large dimensionality, increasing time complexity, and could result in unnecessary noise. Furthermore, too small of a vocabulary could result in insufficient information to classify each document with. The typical approach for selecting the feature extraction vocabulary is to use every term in the corpus. This approach however is problematic as it creates such a large dimensionality (1,970,810 terms in the largest cast), and when scaling this work the time complexity becomes a barrier. To avoid this issue, the CWE-888 data was preprocessed in the same was as mentioned in the previous Section, and the remaining terms were used as the vocabulary (reducing the dimensionality to 2938 terms).

The most simplistic feature extraction method is the Binary Bag-of-Words Frequency (BF) as shown in Equation 4.1. This method only determines if each term in the vocabulary is in the document or not. Equation 4.1 shows that the BF of any term can only be 1 or 0. BF(term) is the binary frequency of term, and f(term) represents the frequency (or number of occurrences) of term in document.

$$BF(term) = \begin{cases} 0 & f(term) = 0 \\ 1 & f(term) > 0 \end{cases} \tag{4.1}$$

The Term Frequency (TF) feature extraction method (as shown in Equation 4.2) retains more information about the document than the BF. Instead of reducing a document into 1's and 0's corresponding to the presence or absence of a term, TF records the frequency (or

number of occurrences) of term in the document. TF(term) is the term frequency of term, and f(term) represents the frequency (or number of occurrences) of term in the document.

$$TF(term) = f(term) \tag{4.2}$$

The Term Frequency-Inverse Document Frequency (TF-IDF) feature extraction method (as shown in Equation 4.3) is an extension of the TF feature extraction method, that weights the importance of a term in a specific document inversely to how often it appears in other documents. This is done to decrease the effect a term which appears in many documents has on the feature vector, as a term which appears in a wide range of documents would contain little discriminatory information. $tfidf(term)$ represents the tfidf score of term in the document, $f(term)$ is the frequency (or number of occurrences) of term in the document, n is the total number of documents, and $N(term)$ is the number of documents that term appears in.

$$tfidf(term) = f(term) * log\frac{n}{N(term)} \tag{4.3}$$

A common variation to these feature extraction methods is to exclude any terms that do not appear a minimum number of times in a document. This minimum frequency is often used to reduce the noise of the dataset, however this work focused on bug reports which often include only one word pertaining to the security aspect of the issue. Therefore, no minimum frequency was set to avoid loosing important information.

4.3 Classifiers

Machine learning classifiers fall into two main categories: supervised and unsupervised [46]. A supervised learning technique is any approach in which the true class of the training data is used to infer a function or model to describe the output from the input data. The supervised learning methods used in this work and discussed below are: Bayesian Network (BN), k-Nearest Neighbor (kNN), Naive Bayes (NB), Naive Bayes Multinomial (NBM), Random Forest (RF), and Support Vector Machine (SVM).

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{4.4}$$

Equation 4.4 presents Bayes' theorem where P(A) and P(B) are the probabilities of observing A and B without regard to each other, $P(A|B)$ is the probability of observing event A given that B is true (referred to as a conditional probability), and $P(B|A)$ is the probability of observing event B given that A is true [47]. This theorem is the basis for the Naive Bayes, Naive Bayes Multinomial, and Bayesian Network classifiers described below.

$$P(X|C) = \Pi_{i=1}^{n} P(X_i|C) \tag{4.5}$$

The Naive Bayes classifier is a result of the unrealistic assumption that strong independence exists between the features, and presented in Equation 4.5, where X is a feature vector and C is a class. Although this assumption is unrealistic, the Naive Bayes classifier is remarkably successful in practice [48]. The implementation of this algorithm in Weka was used [49].

$$P(x|C_k) = \frac{(\sum_i x_i)!}{\Pi_i x_i!} \Pi_i p_{ki}^{xi} \tag{4.6}$$

As an expansion of the Naive Bayes classifier, a multinomial event model is expected correlating to the feature vectors being probabilities of a multinomial distribution [50]. The multinomial expansion on the Naive Bayes Theorem is show in in Equation 4.6 where x is the feature vector, p_i is the probability that even i occurs, and k is the class. In simpler terms, this assumes the input is of a multinomial distribution. This algorithm was implemented using Weka [49].

A Bayesian Network (or Bayes Net) is a probabilistic directed acyclic graphical model that represents a set of random variables and their conditional dependencies [51]. In simpler terms, a Bayesian network can be considered a mechanism for automatically applying Bayes' theorem (as shown in Equation 4.4) to complex problems. A common example is that a Bayesian Network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases. Bayes Net was implemented using Weka [49].

Random Forests operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes of the individual trees [52]. Decision trees are know to overfit, and therefore the Random forest approach helps to correct for this overfitting. Decision trees map observations (branches) about an item to conclusions about the item's target value (leaves). This classifier was implemented using Weka [49].

Support Vector Machine (SVM) is a non-probabilistic binary linear classifier, which maximizes the distance between the decision line (the line separating the two classes) and each of the two classes [53]. Although this is a linear model, it can efficiently perform non-linear classification using a kernel trick which maps the input into a higher dimensionality feature space. This classifier was implemented using Weka [49].

Unsupervised learning is any technique in which algorithms or models are used to infer a function to describe hidden structure from unlabeled data. The unsupervised classifiers used in this work and discussed below are Cosine Similarity and kNN.

k-Nearest Neighbor (kNN) is a classification method in which an input is classified by a majority vote of its closest neighbors, with the input being assigned to the class most common among its k nearest neighbors (where k is a positive integer) [54]. This among the simplest of machine learning algorithms, and is a form of lazy-learning where all computation is deferred until classification. The distance metric used for this classifier was the Euclidean distance as described in Equation 4.7, where $D(a, b)$ represents the Euclidean distance between vectors a and b and n represents the dimensionality of the vectors. This classifier was implemented using Weka [49].

$$D(a, b) = \sqrt{\sum_{i=1}^{n}(b_i - a_i)} \tag{4.7}$$

Equation 4.8 shows the formula used to calculate the cosine similarity between two vectors A and B [55]. While this is simply a distance metric, it can be used to represent the similarity between two documents that are represented in feature vectors. Therefore, the documents can be classified as the class of which they are most "similar." This method was implemented using python [45].

$$similarity(A, B) = \frac{A * B}{||A||||B||} \tag{4.8}$$

4.4 Performance Evaluation

The metrics used for performance evaluation are derived from the confusion matrix shown in Table 4.1 [56]. The true or positive class for this work was chosen to be security related issues, and the false or negative class was chosen to be the non-security issues. The following performance metrics were used: accuracy, precision, recall, probability of false alarm (PFA), F-Score, and G-Score. The equations and description of each metric are given below.

Table 4.1: Performance Measure Confusion Matrix

		Predicted Class Class	
		Security Issue	Non-Security Issue
True Class	Security Issue	True Positive (TP)	False Negative (FN)
	Non-Security Issue	False Positive (FP)	True Negative (TN)

The accuracy as shown in Equation 4.9 describes the total number of correctly classified issues with respect to all issues. This metric is of limited value to this work due to the effect imbalanced data has on it. For example, if only 10% of the issues in a project were security related, then an accuracy of 90% could be obtained simply by labeling all issues as non-security.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.9}$$

The precision as shown in Equation 4.10 describes the total number of correctly classified security issues out of all issues determined to be security related. Again, this measure can be skewed with imbalanced data.

$$Precision = \frac{TP}{TP + FP} \tag{4.10}$$

The recall as shown in Equation 4.11 is one of the most important performance metrics for this work. The recall describes the probability of detecting a security issue. This metric is one of the most important for this work as missing a security issue can lead to vulnerabilities that significantly decrease the software system's integrity.

$$Recall = \frac{TP}{TP + FN} \tag{4.11}$$

The probability of false alarm (PFA) describes the probability of labeling a non-security issue as security. This measure tends to be very high in automated security tasks in an attempt to catch all security issues.

$$Probability\ of\ False\ Alarm\ (PFA) = \frac{FP}{TN + FP} \tag{4.12}$$

The F-Score is the harmonic mean between precision and recall, which describes how well an automated system is able to balance the performance between precision and recall. A system with a higher F-Score usually relates to better performance.

$$F\text{-}Score = 2 * \frac{precision * recall}{precision + recall} \tag{4.13}$$

The G-Score is the harmonic mean between recall and the probability of false alarm. This metric will be highly scrutinized as well as it accounts for the two most important measures in security detection systems.

$$G\text{-}Score = 2 * \frac{recall * (1 - PFA)}{recall + (1 - PFA)} \tag{4.14}$$

When reporting the results from multiclass classification, both the macro-averaged and the weighted average of the performance metrics listed above were used. Macro-averaging treats each class as equally important, whereas the weighted average weights each class with respect to the number of instances it contains. The weighted average is then biased towards the largest or larger class(es), whereas the macro-averaged performance metrics may not show performance which accurately describes the number of correctly classified instances. The weight average and macro-average equations are shown below where i represents the

number of classes, M_i represents the performance metric being averaged for each class, and w_i represents the number of bug reports in each class.

$$Macro\text{-}average = \frac{1}{i}\sum_{i=1}^{i} M_i \tag{4.15}$$

$$Weighted\ Average = \frac{\sum_{i=0}^{n} M_i * w_i}{\sum_{i=0}^{n} w_i} \tag{4.16}$$

4.5 Supervised Learning

As defined in Section 4.3, supervised learning is an approach in which the true class of the training data is used to infer a function or model (train a learner) to describe the output from the input data. This section describes the procedure used to train and test each automated system that could be created from combining one of the feature extraction methods mentioned in Section 4.2, as well as one of the supervised classifiers listed in Section 4.3. Each system will be denoted as (Feature Extraction Method)_(Classifier). For example, if the Term Frequency feature extraction method was used with the Naive Bayes Multinomial (NBM) classifier, this would be denoted as TF_NBM.

Even though each system is made up of a different feature extraction and supervised classifier combination, the process used to train and test each system is the same. Therefore, the remainder of this section will use the term system to refer to all feature extraction and classifier combinations.

Each system will be tested on the corpora from each of the datasets described in Section 4.2. Each corpus must be separated into a training and a testing set in which no document appears in both sets. 10-fold stratified cross validation, as well as 75%, 50%, and 25% percentage splits where used to create four sets of training and testing data per corpus, in order to test each system's performance with respect to amount of training data needed.

After these training and testing sets were obtained, each system was presented each document and label in the training set. The system then used the labels to tune the underlying function or model to the training data, by predicting the label of the input document, and comparing this result to the true label of the input document. The system then updates

the underlying function or model accordingly. After the system has been presented with all training documents, the testing documents are presented to the system. The system generates labels for the testing data based on what it learned from the training data, and the performance is then evaluated by comparing the predicted labels to the true labels of the testing issues.

4.6 Supervised Two Class Classification

Using the procedure mentioned above, testing each system on each corpus as a two-class problem meant simply labeling all documents labeled with a CWE-888 class to "Security Related," and label the rest as "Not Security Related." Each systems ability to distinguish between a security and a non-security bug report was tested by addressing research questions 2, 2a, and 2b addressed. These results are detailed in the next section.

2. Can supervised machine learning algorithms be used to classify software issues as security related or not?

    a) Do some learners preform consistently better than others?

    b) How much data must be set aside for training in order to produce accurate classification results?

4.6.1 Two Class Classification Results

Table 4.2 presents the classification performance for each dataset when using Binary Bag-of-Words feature extraction and each supervised classifier, and 10-fold stratified cross validation. The column corresponding to the classifier that performs the best (with respect to G-Score) for each dataset is in bold. Interestingly, the best performing classifier is different for each dataset. Furthermore, a classifier which performs very well on one dataset, may perform very poorly on another. An example of this is the Bayesian Network: while performing the best on the Ground Mission IV&V Issues dataset, it performed the worst (G-Score of 0) on the Flight Mission Developers Issues dataset. When using the BF feature extraction method,

a Bayesian Network classifier is most effective for the Ground Mission IV&V issue dataset, a Random Forest is most effective for the Flight Mission IV&V dataset, and Naive Bayes is most effective for the Flight Mission Developers Issues dataset.

Table 4.2: Two-Class Classification Performance of BF Feature Vector and all Classifiers Across All Projects

	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Ground Mission IV&V Issues	*Accuracy*	**87.4%**	94.6%	87.2%	88.7%	94.8%	94.3%
	Precision	**37.0%**	65.4%	36.7%	39.4%	80.3%	70.7%
	Recall	**93.4%**	62.5%	93.4%	89.7%	41.9%	42.6%
	PFA	**13.1%**	2.7%	13.3%	11.4%	0.9%	1.5%
	F-Score	**0.530**	0.639	0.527	0.547	0.551	0.532
	G-Score	**0.900**	0.761	0.899	0.891	0.589	0.595
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	BF_RF	*BF_SVM*
Flight Mission IV&V Issues	*Accuracy*	69.9%	76.2%	70.7%	80.1%	**84.0%**	81.4%
	Precision	58.3%	70.4%	59.1%	70.6%	**80.8%**	79.1%
	Recall	94.3%	72.6%	93.0%	88.5%	**80.3%**	74.5%
	PFA	47.1%	21.3%	44.9%	25.8%	**13.3%**	13.8%
	F-Score	0.674	0.715	0.723	0.785	**0.805**	0.767
	G-Score	0.678	0.755	0.692	0.807	**0.834**	0.799
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Flight Mission Developers Issues	*Accuracy*	65.8%	66.9%	**66.9%**	70.1%	69.5%	67.1%
	Precision	65.8%	69.4%	**77.7%**	70.2%	69.9%	74.7%
	Recall	100.0%	89.0%	**69.8%**	94.6%	94.4%	75.7%
	PFA	100.0%	75.5%	**38.6%**	77.2%	78.3%	49.5%
	F-Score	0.794	0.780	**0.735**	0.806	0.803	0.752
	G-Score	0.000	0.384	**0.653**	0.367	0.353	0.606

Table 4.3 presents the classification performance for each dataset when using the Term Frequency (TF) feature extraction method and each supervised classifier. The column corresponding to the classifier that performs the best (with respect to G-Score) for each dataset is in bold. Unlike Table 4.2, the best performing classifier (SVM) was consistent across the Flight Mission IV&V Issues and the Flight Mission Developers Issues datasets. Consistently with Table 4.2 however, a classifier that performs well on one dataset does not imply that

it performs well with another. When using the TF feature extraction method, the Naive Bayes Multinomial classifier is most effective for the Ground Mission IV&V Issues, and the Support Vector Machine classifier is the most effective for both Flight Mission datasets.

Table 4.3: Two-Class Classification Performance of TF Feature Vector and all Classifiers Across All Projects

	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Ground Mission IV&V Issues	*Accuracy*	87.4%	93.5%	85.2%	**87.9%**	94.9%	94.1%
	Precision	37.1%	57.3%	32.0%	**38.0%**	82.6%	66.0%
	Recall	93.4%	60.3%	83.1%	**93.4%**	41.9%	47.1%
	PFA	13.1%	3.7%	14.6%	**12.6%**	0.7%	2.0%
	F-Score	0.531	0.588	0.462	**0.540**	0.556	0.549
	G-Score	0.900	0.742	0.842	**0.903**	0.589	0.636
	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Flight Mission IV&V Issues	*Accuracy*	69.6%	70.9%	75.1%	78.3%	80.4%	**83.8%**
	Precision	57.9%	60.3%	67.8%	67.8%	75.9%	**78.8%**
	Recall	95.5%	86.0%	75.2%	89.8%	76.4%	**82.8%**
	PFA	48.4%	39.6%	24.9%	29.8%	16.9%	**15.6%**
	F-Score	0.721	0.709	0.713	0.773	0.762	**0.807**
	G-Score	0.670	0.710	0.751	0.788	0.796	**0.836**
	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Flight Mission Developers Issues	*Accuracy*	65.8%	61.0%	66.9%	70.6%	70.4%	**72.3%**
	Precision	65.8%	71.7%	75.0%	73.6%	71.0%	**76.8%**
	Recall	100.0%	67.2%	74.6%	86.4%	93.2%	**83.1%**
	PFA	100.0%	51.1%	47.8%	59.8%	73.4%	**48.4%**
	F-Score	0.794	0.694	0.748	0.795	0.806	**0.798**
	G-Score	0.000	0.566	0.614	0.549	0.414	**0.637**

Table 4.4 presents the classification performance for each dataset when using the Term Frequency-Inverse Document Frequency (TF-IDF) feature extraction method and each supervised classifier. The column corresponding to the classifier that performs the best (with respect to G-Score) for each dataset is in bold. Similar to what was seen in Table 4.2, the best performing classifier for each dataset was different. As in Tables 4.2 and 4.3, a classifier that performs well on one dataset does not imply that it performs well with another.

When using the TF-IDF feature extraction method, a Bayesian Network performs best on the Ground Mission IV&V Issues dataset, the Naive Bayes Multinomial Classifier performs best on the Flight Mission IV&V Issues dataset, and the Naive Bayes Classifier works best on the Flight Mission Developers Issues dataset.

Table 4.4: Two-Class Classification Performance of TF-IDF Feature Vector and all Classifiers Across All Projects

	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Ground Mission IV&V Issues	*Accuracy*	**87.6%**	93.9%	86.0%	92.8%	94.0%	90.2%
	Precision	**37.3%**	61.7%	34.0%	90.0%	75.0%	40.7%
	Recall	**91.9%**	54.4%	89.0%	6.6%	33.1%	61.0%
	PFA	**12.8%**	2.8%	14.3%	0.1%	0.9%	7.4%
	F-Score	**0.531**	0.578	0.492	0.123	0.459	0.488
	G-Score	**0.895**	0.698	0.873	0.124	0.496	0.735
	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Flight Mission IV&V Issues	*Accuracy*	70.2%	73.3%	79.3%	82.2%	**82.5%**	73.6%
	Precision	58.4%	61.2%	71.2%	90.1%	**80.0%**	67.9%
	Recall	94.9%	95.5%	83.4%	63.7%	**76.4%**	67.5%
	PFA	47.1%	42.2%	23.6%	4.9%	**13.3%**	22.2%
	F-Score	0.723	0.746	0.768	0.746	**0.782**	0.677
	G-Score	0.679	0.720	0.797	0.763	**0.812**	0.723
	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Flight Mission Developers Issues	*Accuracy*	68.0%	64.9%	**62.3%**	66.0%	70.6%	59.9%
	Precision	48.9%	66.7%	**73.4%**	65.9%	71.2%	73.0%
	Recall	94.4%	93.2%	**66.9%**	100.0%	92.9%	61.9%
	PFA	82.6%	89.7%	**46.7%**	99.5%	72.3%	44.0%
	F-Score	0.795	0.777	**0.700**	0.795	0.806	0.670
	G-Score	0.294	0.185	**0.593**	0.010	0.427	0.588

The previous tables have shown promising results for each dataset. Table 4.5 addressed research question 2b which asked: How much data must be set aside for training in order to maintain accurate results? The Binary bag-of-words feature vector was used along with the Naive Bayes feature vector to address this question: there was no significant difference between any of the feature extraction methods, and the Naive Bayes classifier was the most consistently acceptable performing classier across all datasets. As show below, the Ground Mission IV&V Issues dataset, along with the Flight Mission IV&V Issues dataset both achieved the best performance when using only 25% of the data for training. Furthermore, The Flight Mission Developers Issues dataset performed the best when using only 50% of

the data for training. A system which is able to perform well when using a small amount of data for training is significantly more practical, than one which needs 90% of the data for training (as in any example using 10-fold cross validation).

Table 4.5: Performance of BF_NB on All Projects vs Amount of Training Data

	% of Issues for Training	*10% Stratified Cross Validation*	*75%*	*50%*	*25%*
Ground Mission IV&V Issues	*Accuracy*	**87.2%**	86.3%	85.6%	86.7%
	Precision	**36.7%**	38.9%	34.0%	36.9%
	Recall	**93.4%**	92.5%	94.1%	93.5%
	PFA	**13.3%**	14.3%	15.1%	13.9%
	F-Score	**0.527**	0.548	0.500	0.529
	G-Score	**0.899**	0.890	0.893	0.896
	% of Issues for Training	*10% Stratified Cross Validation*	*75%*	*50%*	*25%*
Flight Mission IV&V Issues	*Accuracy*	70.7%	71.6%	76.4%	**77.3%**
	Precision	59.1%	83.7%	87.5%	**90.5%**
	Recall	93.0%	54.2%	66.7%	**68.3%**
	PFA	44.9%	10.6%	11.6%	**10.1%**
	F-Score	0.723	0.658	0.757	**0.778**
	G-Score	0.692	0.675	0.760	**0.776**
	% of Issues for Training	*10% Stratified Cross Validation*	*75%*	*50%*	*25%*
Flight Mission Developers Issues	*Accuracy*	66.9%	62.7%	**65.1%**	66.0%
	Precision	77.7%	80.3%	**78.5%**	75.9%
	Recall	69.8%	58.9%	**64.2%**	71.1%
	PFA	38.6%	29.5%	**33.3%**	43.8%
	F-Score	0.735	0.680	**0.706**	0.734
	G-Score	0.653	0.642	**0.654**	0.628

4.6.2 Two Class Classification Observations

The two class classification of bug reports has been shown possible with good performance. Furthermore, it was shown that as small as 25% of the data can be used for testing without degrading the classification performance. The feature vector used did not effect the results significantly, and the Naieve Bayes classifier was consistently the best performing, or among the best performing classifiers for all datasets. SVM was another consistently high performing classifier, yet was not as good as Naive Bayes.

The part about the smaller training set can be expanded

4.7 Supervised Multiclass Classification

Each of the corpus as defined in Section 4.2 are labeled for a multiclass classification problem. Each document is labeled with either "Not Security Related" or its corresponding CWE-888 class. With the data in this form, it can be processed as previously described where each document will be classified by the system into one of the CWE-888 classes, or as "Not Security Related." This section addressed research questions 3 and 3a, and the results are detailed in the following section.

3. Can supervised machine learning algorithms be used to classify security issues to specific security classes?

    a) Are some classes harder to predict than others?

4.7.1 Multiclass Classification Results

The following three tables will present the supervised systems performance on the multiclass problem, identifying each issue as its security type. Table 4.6 shows the systems performance when using the BF feature extraction method on each of the datasets for multiclass classification. As seen in the two class results, just because a classifier performs well on one dataset does not mean that it will perform well on another. When using the BF feature extraction method, as well as the weighted average of each individual classes performance,

the Naive Bayes classifier performed the best for both IV&V Issues datasets, and k-Nearest Neighbor performed the best on the Flight Mission Developers Issues Dataset.

Table 4.6: Multiclass Classification Weighted Average Performance of BF Feature Vector and all Classifiers Across All Projects

	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Ground Mission IV&V Issues	*Precision*	93.9%	94.1%	**94.1%**	91.1%	92.1%	92.1%
	Recall	84.9%	93.9%	**86.0%**	91.5%	94.5%	87.0%
	PFA	8.4%	33.3%	**5.1%**	32.8%	60.5%	46.9%
	F-Score	0.892	0.940	**0.899**	0.913	0.933	0.895
	G-Score	0.881	0.780	**0.902**	0.775	0.557	0.659
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Flight Mission IV&V Issues	*Precision*	66.3%	63.3%	**70.2%**	65.2%	67.3%	48.0%
	Recall	50.0%	63.9%	**59.9%**	66.8%	68.6%	35.9%
	PFA	10.3%	28.8%	**9.7%**	34.5%	40.1%	15.7%
	F-Score	0.570	0.636	**0.646**	0.660	0.679	0.411
	G-Score	0.642	0.674	**0.720**	0.661	0.640	0.504
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Flight Mission Developers Issues	*Precision*	12.9%	**30.7%**	41.8%	44.0%	44.7%	26.9%
	Recall	33.8%	**98.5%**	44.8%	47.2%	48.7%	6.3%
	PFA	33.4%	**29.9%**	18.7%	25.2%	23.5%	5.4%
	F-Score	0.187	**0.468**	0.432	0.455	0.466	0.102
	G-Score	0.448	**0.819**	0.578	0.579	0.595	0.118

Table 4.7 shows the systems performance when using the TF feature extraction method on each of the datasets for multiclass classification. As shown in the two class results, just because a classifier performs well on one dataset does not mean that it will perform well on another. When using the TF feature extraction method, along with the weighted average of each individual class's performance, the Naive Bayes classifier performed the best for the Ground Mission IV&V Issues dataset, and the Naive Bayes Multinomial classifier performed the best for both of the Flight Mission datasets.

Table 4.7: Multiclass Classification Weighted Average Performance of TF Feature Vector and all Classifiers Across All Projects

Ground Mission IV&V Issues	*Supervised System*	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
	Precision	93.8%	92.5%	**92.5%**	94.5%	91.9%	90.9%
	Recall	85.2%	92.0%	**80.8%**	88.9%	94.3%	92.3%
	PFA	9.1%	37.4%	**2.7%**	14.7%	62.5%	65.2%
	F-Score	0.893	0.922	**0.863**	0.916	0.931	0.916
	G-Score	0.880	0.745	**0.883**	0.871	0.537	0.505
Flight Mission IV&V Issues	*Supervised System*	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
	Precision	66.1%	67.1%	63.9%	**69.5%**	66.1%	56.5%
	Recall	48.4%	51.8%	59.9%	**67.3%**	68.3%	46.3%
	PFA	10.3%	21.7%	5.6%	**17.9%**	39.5%	19.4%
	F-Score	0.559	0.585	0.618	**0.684**	0.672	0.509
	G-Score	0.629	0.624	0.733	**0.740**	0.642	0.588
Flight Mission Developers Issues	*Supervised System*	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
	Precision	12.6%	23.5%	36.7%	**39.3%**	41.9%	28.7%
	Recall	33.3%	37.2%	33.1%	**46.7%**	46.1%	13.0%
	PFA	33.4%	31.2%	15.0%	**23.4%**	24.7%	8.3%
	F-Score	0.183	0.288	0.348	**0.427**	0.439	0.179
	G-Score	0.444	0.483	0.476	**0.580**	0.572	0.228

Table 4.8 shows the systems performance when using the TF-IDF feature extraction method on each of the datasets for multiclass classification. This table again shows that just because a classifier performs well on one dataset does not imply that it will perfor well on anther. Again, this table shows the best performing classifier being different for each dataset. When using the TF-IDF feature extraction method, along with the weighted average of each individual class's performance, a Bayesian Network performed best for the Ground Mission IV&V Issues dataset, the Naive Bayes classifier performed the best for the Flight Mission IV&V Issues dataset, and the Random Forest classifier performed the best for the Flight Mission Developers Issues dataset.

Table 4.8: Multiclass Classification Weighted Average Performance of TF-IDF Feature Vector and all Classifiers Across All Projects

Ground Mission IV&V Issues	*Supervised System*	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
	Precision	**94.1%**	92.6%	93.1%	85.3%	91.0%	92.5%
	Recall	**85.5%**	93.1%	76.8%	92.4%	94.0%	43.7%
	PFA	**9.0%**	42.2%	11.8%	92.4%	67.3%	7.8%
	F-Score	**0.896**	0.928	0.842	0.887	0.925	0.594
	G-Score	**0.882**	0.713	0.821	0.140	0.485	0.593
Flight Mission IV&V Issues	*Supervised System*	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
	Precision	67.5%	66.8%	65.7%	34.7%	**69.9%**	64.7%
	Recall	50.5%	47.1%	63.6%	85.9%	**68.8%**	31.7%
	PFA	9.9%	11.6%	19.2%	58.9%	**7.2%**	5.3%
	F-Score	0.578	0.552	0.646	0.494	**0.693**	0.426
	G-Score	0.647	0.615	0.712	0.556	**0.790**	0.475
Flight Mission Developers Issues	*Supervised System*	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
	Precision	13.4%	31.8%	33.4%	49.2%	**43.7%**	28.8%
	Recall	33.8%	35.7%	31.2%	40.9%	**46.7%**	10.8%
	PFA	32.7%	31.2%	5.4%	29.3%	**24.7%**	7.0%
	F-Score	0.192	0.336	0.323	0.447	**0.452**	0.157
	G-Score	0.450	0.470	0.469	0.518	**0.576**	0.194

The previous three tables have detailed the supervised systems performance on the multiclass problem (each issue as its corresponding security type), based on the weighted average of the performance metrics for each individual classes. The following three tables show the supervised systems performance on the multiclass problem, but using the macro-averaged performance metrics of each individual classes. Table 4.9 shows the systems macro-averaged performance when using the BF feature extraction method on each of the datasets for multiclass classification. Consistently with all other results presented thus far, just because a classifier performs well on one dataset does not mean that it will perform well on another. When using the BF feature extraction method and macro-averaged performance metrics for multiclass classification, the best performing classifier for the Ground Mission IV&V Issues dataset was a Bayesian Network, the best performing classifier for both Flight Missions was Naive Bayes.

Table 4.9: Multiclass Classification Macro-Averaged Performance of BF Feature Vector and all Classifiers Across All Projects

	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Ground Mission IV&V Issues	*Accuracy*	84.9%	84.9%	**86.0%**	91.5%	94.5%	87.0%
	Precision	15.5%	21.9%	**54.7%**	10.4%	33.3%	24.6%
	Recall	26.1%	15.1%	**35.6%**	14.0%	20.5%	18.2%
	PFA	1.8%	4.6%	**1.5%**	3.2%	5.1%	4.6%
	F-Score	0.194	0.179	**0.431**	0.119	0.254	0.209
	G-Score	0.41	0.261	**0.523**	0.245	0.337	0.306
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Flight Mission IV&V Issues	*Accuracy*	50.0%	63.9%	**59.9%**	66.8%	68.6%	35.9%
	Precision	25.2%	28.8%	**31.1%**	38.6%	39.0%	12.6%
	Recall	27.1%	28.1%	**35.3%**	24.0%	25.7%	18.2%
	PFA	6.0%	6.5%	**5.0%**	6.8%	7.2%	8.0%
	F-Score	0.261	0.284	**0.331**	0.296	0.310	0.149
	G-Score	0.421	0.432	**0.515**	0.382	0.403	0.304
	Supervised System	*BF_BN*	*BF_kNN*	*BF_NB*	*BF_NBM*	*BF_RF*	*BF_SVM*
Flight Mission Developers Issues	*Accuracy*	33.8%	38.5%	**44.8%**	47.2%	48.7%	6.3%
	Precision	3.4%	7.4%	**20.6%**	15.7%	21.4%	7.4%
	Recall	6.7%	7.5%	**17.1%**	11.8%	13.5%	8.0%
	PFA	6.2%	5.7%	**4.6%**	4.9%	4.7%	6.2%
	F-Score	0.045	0.074	**0.187**	0.135	0.166	0.077
	G-Score	0.125	0.139	**0.290**	0.210	0.236	0.147

Table 4.10 shows the systems macro-averaged performance when using the TF feature extraction method on each of the datasets for multiclass classification. Consistently with all other results presented thus far, just because a classifier performs well on one dataset does not mean that it will perform well on another. This table shows the same classifier performing the best across all datasets, which has not yet been observed. The best performing classifier for all datasets was Naive Bayes.

Table 4.10: Multiclass Classification Macro-Averaged Performance of TF Feature Vector and all Classifiers Across All Projects

	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Ground Mission IV&V Issues	*Accuracy*	85.2%	92.0%	**80.8%**	88.9%	94.3%	92.3%
	Precision	15.5%	27.0%	**15.3%**	33.3%	34.1%	30.5%
	Recall	26.8%	25.8%	**35.3%**	21.2%	19.4%	18.3%
	PFA	1.9%	3.5%	**2.7%**	2.0%	5.2%	5.6%
	F-Score	0.196	0.264	**0.213**	0.259	0.247	0.229
	G-Score	0.421	0.407	**0.518**	0.349	0.322	0.307
	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Flight Mission IV&V Issues	*Accuracy*	48.4%	51.8%	**59.9%**	67.3%	68.3%	46.3%
	Precision	24.1%	31.2%	**32.4%**	42.0%	37.9%	25.2%
	Recall	25.5%	23.2%	**34.6%**	31.8%	24.9%	22.9%
	PFA	6.2%	7.0%	**5.6%**	5.1%	7.1%	7.3%
	F-Score	0.248	0.266	**0.335**	0.362	0.301	0.240
	G-Score	0.401	0.371	**0.506**	0.476	0.393	0.367
	Supervised System	*TF_BN*	*TF_kNN*	*TF_NB*	*TF_NBM*	*TF_RF*	*TF_SVM*
Flight Mission Developers Issues	*Accuracy*	33.3%	37.2%	**33.1%**	46.7%	46.1%	13.0%
	Precision	3.2%	4.8%	**13.3%**	13.1%	20.6%	8.6%
	Recall	6.6%	7.1%	**14.7%**	12.4%	13.0%	10.4%
	PFA	6.3%	5.9%	**5.1%**	4.8%	4.9%	6.0%
	F-Score	0.043	0.057	**0.140**	0.127	0.159	0.094
	G-Score	0.123	0.132	**0.255**	0.219	0.229	0.187

Table 4.11 shows the systems macro-averaged performance when using the TF-IDF feature extraction method on each of the datasets for multiclass classification. Consistently with all other results presented thus far, just because a classifier performs well on one dataset does not mean that it will perform well on another. Just as with Table 4.10, Table 4.11 shows the same classifier performing the best across all datasets, which also happens to be the same classifier. The best performing classifier for all datasets was Naive Bayes.

Table 4.11: Multiclass Classification Macro-Averaged Performance of TF-IDF Feature Vector and all Classifiers Across All Projects

	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Ground Mission IV&V Issues	*Accuracy*	85.5%	93.2%	**76.8%**	92.4%	94.0%	43.7%
	Precision	16.6%	30.0%	**18.0%**	7.1%	26.5%	11.9%
	Recall	26.2%	26.9%	**32.6%**	7.7%	17.3%	20.4%
	PFA	1.8%	3.8%	**2.7%**	7.7%	5.6%	4.9%
	F-Score	0.203	0.284	**0.232**	0.074	0.209	0.150
	G-Score	0.414	0.420	**0.488**	0.142	0.292	0.336
	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Flight Mission IV&V Issues	*Accuracy*	50.5%	47.1%	**63.6%**	58.9%	68.8%	31.7%
	Precision	25.8%	25.1%	**32.7%**	5.9%	41.0%	24.5%
	Recall	26.3%	24.5%	**31.5%**	10.0%	25.1%	24.6%
	PFA	5.9%	6.4%	**5.6%**	10.0%	7.2%	7.4%
	F-Score	0.260	0.248	**0.321**	0.074	0.311	0.245
	G-Score	0.411	0.388	**0.472**	0.180	0.395	0.389
	Supervised System	*TFIDF_BN*	*TFIDF_kNN*	*TFIDF_NB*	*TFIDF_NBM*	*TFIDF_RF*	*TFIDF_SVM*
Flight Mission Developers Issues	*Accuracy*	33.8%	35.7%	**31.2%**	40.9%	46.7%	10.8%
	Precision	3.9%	12.2%	**13.2%**	18.1%	21.2%	11.4%
	Recall	7.4%	7.2%	**14.9%**	8.3%	1.3%	7.7%
	PFA	6.2%	6.0%	**5.4%**	5.5%	4.9%	6.0%
	F-Score	0.051	0.091	**0.140**	0.114	0.024	0.092
	G-Score	0.137	0.134	**0.257**	0.153	0.025	0.142

Figures 4.1, 4.2, and 4.3 show heatmaps corresponding to a typical confusion matrix for each dataset. The density of each cell is represented by shades of grey and represents the percentage of each true class assigned to each predicted class. The main diagonal corresponds to correct classification assignments, and therefore a good assignment would consist of dark squares along the diagonal, and white everywhere else. Furthermore, a dark vertical line would represent everything being classified into a single class, and therefore poor system performance.

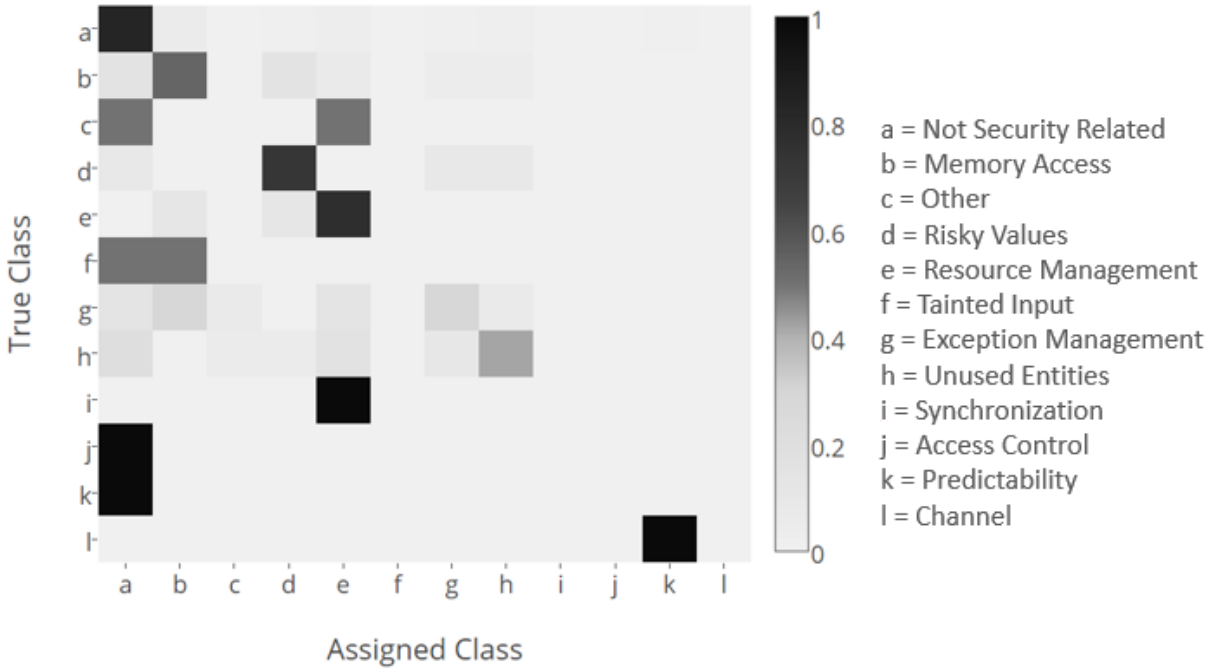

Figure 4.1: Multiclass Classification Heatmap of Ground Mission IV&V Issues dataset using TF_NB

Figure 4.1 shows five columns with multiple dark squares in them. This corresponds to a large portion of bug reports being classified as the classes corresponding to those columns. These highly assigned classes are "Not Security Related," "Memory Management," "Risky Values," "Resource Management," and "Predictability." Interestingly enough, all of these highly assigned classes are the largest classes found in each dataset, with the exception of "Predictability." The best performing classes are "Not Security Related," "Memory Access," "Risky Values," "Resource Management," and "Unused Entities." These are noted as best performing due to the dark squares along the diagonal, representing the majority of all issues that belong to those classes actually being assigned to those classes.

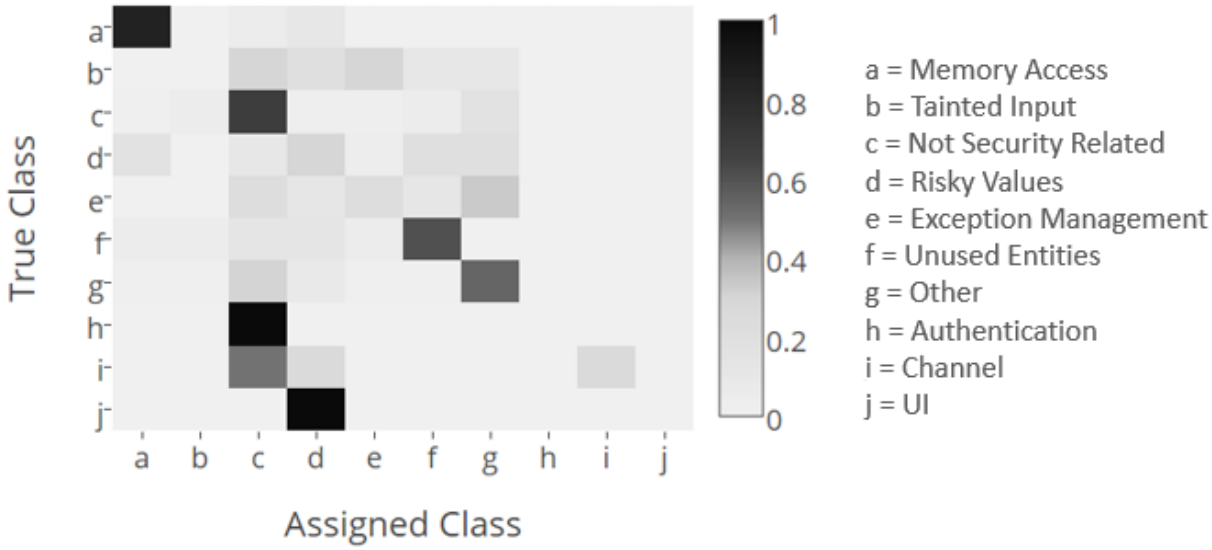

Figure 4.2: Multiclass Classification Heatmap of Flight Mission IV&V Issues dataset using TF_NB

For the same reasoning as described for Figure 4.1, Figure 4.2 shows the majority of all issues were assigned to "Memory Access," "Not Security Related," and "Risky Values." The best performing classes were "Memory Access," "Not Security Related," "Unused Entities," and "Other."

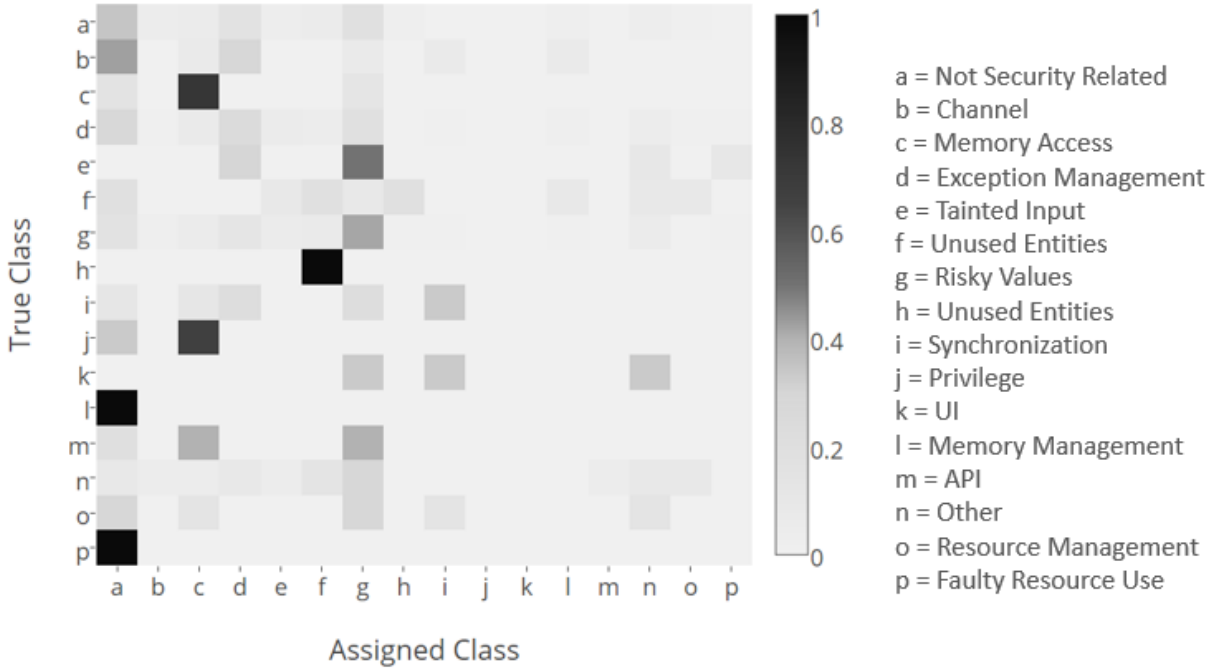

Figure 4.3: Multiclass Classification Heatmap of Flight Mission Developers Issues dataset using TF_NB

For the same reasoning as described for Figure 4.1, Figure 4.3 shows the majority of all issues were assigned to "Not Security Related," "Memory Access," and "Risky Values." The best performing classes were "Memory Access" and "Risky Values."

4.7.2 Multiclass Classification Observations

All performance metrics were presented macro-averaged as well as weighted averaged. The difference in these can be summarized as the macro-average detailing the classification performance with respect to each class, no matter the size of the class, whereas the weighted average details the classification performance with respect to the majority of the issues. All classifiers performed well with respect to the weighted average except for SVM, which significantly underperformed other classifiers in several cases. With respect to the macro-averages however, Naive Bayes outperformed all classifiers in all situations.

The best performing security classes most likely perform well because they contain class specific, well known terminology. The class of "Memory Access" (one of the best performing classes) revolves around problems with pointers and buffers. These are well known and

commonly used terms. However, a class such as "Synchronization" is more centered around the order and process in which things occur; such a class has little class specific terminology that is not as well known as something such as 'pointer,' and therefore is more difficult to accurately predict.

4.8 Unsupervised One Class Problem

The previous section (Section 4.5) defined the process in which supervised systems are trained and tested. Supervised systems rely heavily on labeled data as defined in the definition of supervised learning, and shown in the previous section. An enormous amount of time and effort is needed to manually label each issue (or a significant portion of all issues) properly for use in any of the aforementioned systems. Furthermore, there is a very high likelihood that not all classes (especially in the case of multiclass classification) will be amply defined, or even present in the training set. Obviously, if a class is never defined to a classifier, it is impossible for that classifier to correctly classify an issue of the never defined class. The results shown multiple times throughout Chapter 3 proves that this situation exists for every project analyzed.

In order to avoid this time consuming and costly requirement of manual labeling as well as guarantee that all classes have been properly defined and amply presented to a classifier, this section presents a distance metric based approach to classifying issues. The CWE-888 view defined in Section 3.2 was used to define the security type of each issue during labeling as defined in Section 3.3. Furthermore, all CWE's in the CWE-888 view (from here on out referred to as the CWE-888 data) describe the features of each class, and can be used for one class classification.

This approach is one of anomaly detection. Anomaly detection refers to the problem of finding patterns in data that deviate from a normal [43]. The CWE-888 data was used to define this normal. Specifically, each CWE in the CWE-888 view was processed just as each issue was in Sections 4.1 and 4.2, which put this data in the same form as the documents in the corpora. The definition of the normal will be covered in Section 4.8.2.

Now that the data is in a format consistent with the documents (issues), the cosine

similarity distance measure can be used to determine if a document conforms to the normal or not. The cosine similarity and Euclidean distance simply measures the distance (angle) between the normal and the document, and if the distance is greater than a threshold then the issue is said to deviate from the normal, otherwise it does not. Using the Euclidean distance in this sense is essentially an application of the kNN algorithm, with the only difference being that each instance is classified based on its distance from a specified neighbor. The selection of this threshold will be discussed in Section 4.8.1. This section addresses research question 4:

4. Can unsupervised machine learning algorithms be used to classify software issues as security related or not?

4.8.1 Defining a threshold

In many cases, defining a threshold is more of an art than a scientific method. However, [57] described a method which was employed in this work. The threshold was selected using the following steps:

1. Separate the data into three sets: training, validation, and testing
2. Test a wide range of thresholds on the validation data
3. Select the threshold which gives the best performance on the validation data, for use on the testing data

As mentioned in the first step, three subsets of data are needed. The training data will be the CWE-888 data, however the testing and validation data must originate from the corpora. Therefore, each corpus was separated into two subsets using 2-fold cross-validation, where one fold was used as the validation set, and the remaining as the testing set. Furthermore, we selected the G-Score as the metric to base the selection of the threshold on as maximizing either the recall or probability of false alarm could easily lead to everything being classified to a single class. The G-Score incorporates both the recall and probability of false alarm into one metric, and because the recall and probability of false alarm are the metrics we are attempting to maximize, the G-Score was the obvious choice.

4.8.2 One Class Classification

Using the approach mentioned in Section 4.8, the CWE-888 data was defined as the normal. Following this, the corpus for each dataset was separated into a validation and a testing set using 2-fold stratified cross validation. The cosine similarity was used to quantify the distance between each validation document and the normal. A wide range of thresholds were tested, and the one maximizing the performance on the validation dataset was chosen. The cosine similarity is then used in combination with the selected threshold to classify the testing data, with a document conforming to the normal being security related, and a document not conforming to the normal being non-security related.

4.8.3 Unsupervised Classification Results

This section details the results obtained from the methodology described in Section 4.8.2. The performance of all systems using BF feature extraction and TF feature extraction have been very similar thus far. The one class problem was only ran with the TF and TF-IDF feature extraction methods. Figure 4.12 shows the one class performance across all datasets using cosine similarity. The best performance among each metric is shown in bold, and the threshold selected from a validation set is also shown for each test. The highest G-Score obtained using the TF-IDF feature extraction method in combination with the cosine similarity distance metric was on the Ground Mission IV&V Issues dataset. Although this method does not perform as well as the best supervised methods, it obtains results very similar to what is seen among the supervised classifiers.

Table 4.12: One Class Performance Across All Projects using Cosine Similarity

Dataset	*Ground Mission IV&V Issues*		*Flight Mission IV&V Issues*		*Flight Mission Developers Issues*	
Feature Extraction Method	*TF*	*TF-IDF*	*TF*	*TF-IDF*	*TF*	*TF-IDF*
Selected Threshold	0.286	0.263	0.216	0.235	0.260	0.220
Accuracy	64.3%	**73.0%**	67.8%	49.2%	55.4%	51.7%
Precision	15.0%	17.7%	58.1%	41.2%	**69.3%**	65.9%
Recall	**78.7%**	69.9%	77.7%	55.4%	57.9%	55.1%
PFA	36.9%	26.7%	39.1%	**55.1%**	49.4%	54.9%
F-Score	0.252	0.283	**0.665**	0.473	0.631	0.600
G-Score	0.700	**0.715**	0.683	0.496	0.540	0.496

Table 4.13 shows the systems performance when using the Euclidean distance as the distance measure for the one-class problem. The best performance among each metric is shown in bold, and the threshold selected from a validation set is also shown for each test. The highest G-Score obtained using the TF-IDF feature vector in combination with the Euclidean distance metric was on the Ground Mission IV&V Issues dataset. Although this method does not perform as well as the best supervised methods, it obtains results very similar to what is seen among the supervised classifiers.

Table 4.13: One Class Performance Across All Projects using Euclidean Distance

Dataset	*Ground Mission IV&V Issues*		*Flight Mission IV&V Issues*		*Flight Mission Developer Issues*	
Feature Extraction Method	*TF*	*TF-IDF*	*TF*	*TF-IDF*	*TF*	*TF-IDF*
Selected Threshold	8.000	1.214	8.770	1.237	6.708	1.248
Accuracy	68.3%	**73.0%**	66.0%	49.0%	51.1%	51.5%
Precision	10.4%	17.8%	60.0%	41.0%	**66.9%**	65.8%
Recall	41.2%	**69.9%**	51.6%	54.8%	50.8%	54.8%
PFA	29.4%	26.7%	**24.0%**	55.1%	48.4%	54.8%
F-Score	0.166	0.283	0.555	0.469	0.578	**0.598**
G-Score	0.520	**0.715**	0.615	0.493	0.512	0.495

Interestingly, the results in Table 4.13 for the classification of the Ground Mission IV&V Issues dataset using the TF-IDF feature vector was exactly the same as the results seen in Table 4.12. The difference in performance between the cosine similarity and Euclidean distance metric are marginal elsewhere as well. The only case where the G-Score differs by more than 0.07 is in the case of the Ground Mission IV&V Issues dataset and the TF feature vector, where the cosine similarity obtained a G-Score of 0.700 and the Euclidean distance obtained a G-Score of 0.520.

4.8.4 Unsupervised Observations and Comparisons with Supervised Techniques

The difference in the classification performance between the cosine similarity and the euclidean distance were marginal. The cosine similarity thresholds did not very significantly across different datasets or different feature vectors, however the thresholds used for the euclidean distance varied significantly across feature vectors, but not across datasets. In all cases, the cosine similarity distance measure obtained equal or higher recalls than the euclidean distance, but the Euclidean distance obtained an equal or better probability of false alarm than the cosine similarity. Furthermore, the F-Score was equal between the two

distance metrics, or higher for the cosine similarity in all cases.

The best unsupervised classification results (G-Score of 0.715) were not as good as the best supervised classification results (G-Score of 0.903), however, the best unsupervised classification results are comparable to the average supervised classification results. Although these two methods are comparable, in general the supervised classifiers outperformed the unsupervised classifiers on all performance metrics.

4.9 Threats to Validity

Many issues arose during the automated classification of bug reports. Some instances arose when an issue could be correctly classified into multiple CWE-888 classes, and therefore could generate two correct results. This has no clear solution and was solved in the case of manual labeling by selecting the most relevant of the possible classes, which possibly could be counting an assigned issue as incorrect, when in fact the classification was accurate. This could have a significant negative impact on the results to the automated classification with regards to multiclass classification.

Each dataset contained a small amount (15% or less) if issues that did not contain sufficient information for classification. The assumption was made that due to the low level of information included in the issue, more information must be provided before the issue is solved. This would lead to either an analyst or developer looking further into the issue and providing more detail, or another issue being opened entirely which more accurately and completely describes the issue. Because of this, an issue which did not contain the information necessary for classification was labeled as "Not Security Related," and most likely does not affect the automated classification significantly.

A troubling factor faced throughout the automated classification was the amount of noise contained within each issue. Remembering that the purpose of a bug tracking system is to detail issues with the system, the vast majority of issues are focused on finding, describing, and fixing a bug. Therefore, the security impact or traits of an issue are most often a small detail within each issue, or not even present. When using traditional feature extraction methods, the most often goal of the following classification is to detect the topic of the

document under concern; however in this case, the goal is to ascertain the security relevance of the issue instead of the topic. Because of this, each feature vector contained a significant amount of noise, often with only a very few terms (as small as one) relating to the security aspect. The attempt of solving this in this work was using the vocabulary extracted from the CWE-888 list, however more delicate solutions could lead to significantly better performance.

Another influencing factor is the terminology used. If the data being classified used terminology to denote security issues that does not exist in the CWE list, then those terms are not being extracted, and therefore can have no effect on the classification. Although this could not be verified, after the manual classification of each issue in Chapter 3, this did not appear to be the case.

Quantifying the performance of any classification system is a difficult task, as many different performance metrics exist which all bring unique benefits and costs. This thesis utilized the G-Score as the main classification method as it is able to quantify the recall and probability of false alarm in a single number, as well as includes many other performance metrics in an attempt to provide the entire picture of the performance of the system. This however, is also complicated for multiclass classification, due to the difference in macro-averaging and the weighted averages of each performance metric. To help avoid experimental bias in this regard, many performance metrics were provided.

Arguably the largest threat to validity is the quality of the provided data. While these NASA bug tracking systems are well maintained, our work attempts to leverage this issue tracking data in a way in which it was not originally meant to be used. Therefore, sufficient information to perform the tasks at hand may not be provided, but other secondary patterns may be used by the classifiers in an attempt to make sinse of, and classify each issue. Whether each issue has sufficient information regarding the security implication of an issue depends on the security knowledge of the developer or analyst entering the issue into the system.

4.10 Automated Classification Conclusion

Table 4.14 presents a comparison between this project and related works. The first row in this table represents the work presented in this thesis, whereas each of the other rows

represent a related work.

Table 4.14: Comparison with Related Works

Comparison with Related Work						
Paper	Purpose	Dataset	Feature Vectors	Classifiers	Performance Metrics	Best Performance
This work	Separate bugs into security categories	NASA	TF, TF-IDF, Binary Bag of Words	Cosine Similarity, Naive Bayes, Naive Bayes Multinomial, Random Forest, k-Nearest Neighbor, Support Vector Machine, Bayes Net	Precision, Accuracy, Recall, Probability of False Alarm, F-Score, G-Score	Precision up to 94%, Recall up to 96%, F-Score up to 85%, G-Score up to 90%
[4]	Separate bugs based on security impact	"the Bugzilla repository of bug reports"	TF-IDF	"ector space model", Naive Bayes	Accuracy and Precision	Success Rate: 95.69, Precision Rate: 93.19
[5]	Identify Security issues within an issue tracking system (HIB - Hidden Impact Bugs)	Redhat and Linux Kernel	TF	NB, NBM, Decision Tree	TP Rate (precision), Bayesian Detection (recall)	Bayesian Detection Rate: 0.4, TP Rate: 0.28, TN Rate: 0.99
[6]	Identify security bug reports which were mislabeled as non security bug reports	Cisco Projects	TF	SVM	Accuracy, Precision, Recall	Accuracy of .87, precision of .85, and recall of .88; however this was compared to the output from the static code analysis tool Fortify
[7]	Find security issues which were misclassified as non-security	MySQL	Set known strings to risk values	Get total risk of an issue from feature extraction and calculate which are security problems	N/A	Claim a 657 to 772% increase in the number of vulnerabilities for the MySQL project.
[8]	Classify and categorize vulnerabilities according to their security types	Firefox	CVSS Scores	Bayesian Net	N/A	N/A
[21]	Determine which components of a project are likely to contain vulnerabilities	20 Android Applications	TF	Naive Bayes or Random Forest	Precision, Recall	Precision with Naive Bayes: 0.62 - 1, Recall with Naive Bayes: .32-.92, Precision with random Forest: .59-1, Recall of Random Forest: .24-1

[24]	Determine if bug is a duplicate of one already in the issue tracking system	Mozilla Project	TF + custom term weighting	Cosine Similarity	N/A	8% of duplicate bug reports were filtered out
[25]	Predict the severity of bug	Eclypse and GNOME	TF-IDF	Naive Bayes, Naive Bayes Multinomial, K-Nearest Neighbor, Support Vector Machine	ROC	Area under ROC curve of 0.93
[26]	Classify each bug into corrective maintenance or other kind of activity	Eclipse, Mozilla, JBoss	TF	Naive Bayes, Logistic Regression, Alternate Decision Trees	Precision, Recall	Precision between .64 and .98, Recall between .33 and .97, correct decision rate from .39 to .82
[27]	Classify issue as bug or request using fuzzy logic	HTTPClient, Jackrabbit, Lucene	Membership score based on term frequency	Fuzzy Logic	Precision, Recall, Accuracy, F-Measure	accuracies, marginally better than other classification methods
[28]	Automatic prediction of different bug types using KNN and Naive bayes.	FIT 4 - Multiple Telecom and Banking projects	TF	Naive Bayes and k-Nearest Neighbor	Precision, Recall	Recall from .72 to .91, Precision from .73 to .79
[29]	Automatically route bugs to the appropriate developer for solution	Eclipse Mylyn, Mozilla	TF-IDF, LDA	SVM, KL	Recall, Divergence	Recall similar to those found previously, with better consistency

This chapter addressed the following research questions:

2. Can supervised machine learning algorithms be used to classify software issues as security related or not?

    a) Do some learners preform consistently better than others?

    b) How much data must be set aside for training in order to produce accurate classification results?

3. Can supervised machine learning algorithms be used to classify security issues to specific security classes?

    a) Are some classes harder to predict than others?

4. Can unsupervised machine learning algorithms be used to classify software issues as security related or not?

5. How does the performance of supervised and unsupervised machine learning algorithms compare when classifying software bug reports?

Can supervised machine learning algorithms be used to classify software issues as security related or not? This work has shown that accurate classification results can be obtained using multiple feature extraction methods, and multiple classifiers on each of the datasets. The level of performance however, does depend on the dataset. A G-Score of 0.903 was obtained on the best performing dataset, whereas the worst performing dataset achieved a G-Score of only 0.653.

Do some learners preform consistently better than others? The results showed that this is the case, but in a very unpredictable manor. Some learners do better than others, but the best performing classifier was different depending not only on the feature extraction method, but also on the dataset. In general however, the Naive Bayes classifier was consistently among the best performers.

How much data must be set aside for training in order to produce accurate classification results? This research has shown that the systems performance using only 25% of the data for training a Naive Bayes classifier is just as effective as using 10-fold cross validation.

Can supervised machine learning algorithms be used to classify security issues to specific security classes? The macro-averaged performance metrics were used to evaluate the multi-class performance, and accurate classification results were achieved for some of the classes. A larger sample size could possibly improve these results, as several classes had a very small number of instances (less than 3). Furthermore, the dataset quality continued to effect the classification results, with the same datasets performing best for the two-class classification performing best for the multiclass classification.

Are some classes harder to predict than others? Most classes are hard to detect. However, the dominating classes for each dataset were the best performing classes for each corresponding dataset. This could imply that not enough data is provided for the under performing classes. A more likely explanation however is that the best performing security classes most likely perform well because they contain class specific, well known terminology. The class of "Memory Access" (one of the best performing classes) revolves around problems with point-

ers and buffers. These are well known and commonly used terms. However, a class such as "Synchronization" is more centered around the order and process in which things occur; such a class has little class specific terminology that is not as well known as something such as 'pointer,' and therefore is more difficult to accurately predict.

Can unsupervised machine learning algorithms be used to classify software issues as security related or not? Incorporating the CWE list into an anomaly detection problem allowed for exactly this. The CWE list can be used in conjunction with unsupervised machine learning algorithms to classify software issues as security related or not with performance comparable to supervised methods. The best unsupervised classification results (G-Score of 0.715) were not as good as the best supervised classification results (G-Score of 0.903), however, the best unsupervised classification results are comparable to the average supervised classification results.

How does the performance of supervised and unsupervised machine learning algorithms compare when classifying software bug reports? In most cases, supervised machine learning algorithms perform better than unsupervised; however the best performing unsupervised method's performance is comparable to the supervised methods. Although these two methods are comparable, in general the supervised classifiers outperformed the unsupervised classifiers on all performance metrics.

Chapter 5

Conclusion

This thesis had two main research goals: (1) to explore the distribution and characteristics of security vulnerabilities based on the information provided in bug tracking systems and (2) to develop data analytics approaches for automatic classification of bug reports as security or non-security related. A vulnerability profile was created for three NASA datasets, showing the prominent vulnerability types as well as how those vulnerabilities trend across projects and mission types. Common supervised machine learning algorithms, as well as a novel unsupervised machine learning approach were used to classify vulnerabilities as security or non-security related.

The vulnerability profile revealed that the majority of software vulnerabilities belong only to a small number of types. Specifically, 87% or more of all issues in the analyzed projects fall under the vulnerability type of "Exception Management," "Memory Access," "Other," "Risky Values," or "Unused Entities."

The supervised classifiers of Naive Bayes, Naive Bayes Multinomial, Bayesian Network, k-Nearest Neighbor, Random Forest, and Support Vector Machine were used to distinguish between security and non-security bug reports as a two class problem. Furthermore, each of these classifiers was tested in combination with three feature extraction methods: Binary Bag-of-Words, Term Frequency, and Term Frequency-Inverse Document Frequency. The classification performance of each dataset depended upon the feature vector and classifier used, bug no significant performance difference was seen between feature vectors. The performance of each classifier varied greatly between datasets, however, the Naive Bayes and

SVM classifiers were always among the best performing. All other classifiers performed very poorly on at least one dataset. The Naive Bayes classifier coupled with the Binary Bag-of-Words feature vector was shown to achieve the same level of performance when using only 25% of the data to train on, as when using 10-fold cross validation.

The classifiers and feature vectors mentioned in the previous paragraph were also evaluated on their ability to differentiate between different security types. This performance was evaluated using both the macro-averaged performance metrics as well as the weighted averaged performance metrics. The weighted averaged performance metrics weighted each class with respect to how many issues it contained, with larger classes carrying more weight. With respect to these performance metrics, the best performing classifiers were very similar to the best performing classifiers of the two-class problem. The macro-averaged performance metrics placed equal weight on every class, which significantly changed the performance metrics as several classes with very few (5 or less) instances exist in all datasets. With respect to the macro-averaged performance metrics, the Naive Bayes classifier out preformed all other classifiers in all cases. While many classes did not perform well, the best performing classes seemed to be those with commonly known, class specific terminology.

In an attempt to not only remove the manual labeling constraint of the supervised learning approach, but to also ensure a complete definition of security types, a novel unsupervised machine learning approach was developed. This approach was one of anomaly detection which used the CWE-888 data to define a normal. The cosine similarity and Euclidean distance measures were used to determine if a bug report deviated from the normal (making it non-security) or not (security). These distance measures achieved very similar classification performance, with the cosine similarity narrowly outperforming the Euclidean distance. An interesting observation however is the selected threshold for the cosine similarity varied only slightly between datasets and feature vectors, where the Euclidean distance threshold changed drastically depending on the feature vector. While this approach performed well, it was not as effective as the supervised method, achieving a G-Score of only 0.715 where the best supervised approach achieved a G-Score of 0.903.

To explore the generalizability of vulnerability profiles, open source empirical studies should be preformed. This could easily be performed on bug tracking systems such as

those made available by RedHat or MySQL. Furthermore, the vulnerability profile could be automated for any issue tracking system in which the issues are labeled with a corresponding CWE-888 tag, or a tag that could be easily linked to the CWE-888 classification schema.

An interesting alternative route for this project would be utilizing a custom built neural network for both the feature extraction and classification. This could be done by creating either a convoluted or recurrent neural network which would convolve over the text and output the class. The class could be either security related or not security related, or could even be expanded to do multiclass classification with the CWE-888 structure. This could be a good approach because often times as seen in this project, automated classification is hard due to the high level of noise, and therefore the feature extraction is of utmost importance. A properly built and sufficiently large neural network would have the opportunity to preform its own feature extraction, and assuming a large enough dataset could most likely become very accurate. The size of the dataset is most likely not an issue either however due to the large amount of open source data available.

Another, perhaps more interesting approach would be to expand the anomaly detection to a multiclass problem. This could be done by treating each class of the multiclass problem as a one-class anomaly detection problem. Then select a threshold for each class with a validation set as described in Section 4.8.1. Then when classifying the testing set, run the test data against all classes as a one-class anomaly detection problem (as was done in this work), and if the issue falls under the threshold for any class, assign that issue to that class. In the even that an issue falls under multiple classes, then select the class in which the issue falls under the threshold by the most significant amount. This would most likely be done by dividing the similarity assigned by the distance metric by the threshold, and assigning the issue to the class in which this number is the lowest.

References

[1] M. Hamill and K. Goseva-Popstojanova, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11219-014-9235-5

[2] ——, "Exploring the missing link: an empirical study of software fixes," *Software Testing, Verification and Reliability*, vol. 24, no. 8, pp. 684–705, 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1518

[3] ——, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, July 2009.

[4] D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, Feb 2014, pp. 294–299.

[5] D. Wijayasekara, M. Manic, and M. McQueen, "Vulnerability identification and classification via text mining bug databases," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, Oct 2014, pp. 3612–3618.

[6] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 11–20.

[7] J. L. Wright, J. W. Larsen, and M. McQueen, "Estimating software vulnerabilities: A case study based on the misclassification of bugs in mysql server," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, Sept 2013, pp. 72–81.

[8] J. A. Wang and M. Guo, "Vulnerability categorization using bayesian networks," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '10. New York, NY, USA: ACM, 2010, pp. 29:1–29:4. [Online]. Available: http://doi.acm.org/10.1145/1852666.1852699

[9] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[10] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, Nov 2010, pp. 249–258.

[11] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 447–456.

[12] F. Frattini, R. Ghosh, M. Cinque, A. Rindos, and K. S. Trivedi, "Analysis of bugs in apache virtual computing lab," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–6.

[13] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–8.

[14] ——, "The nature of the times to flight software failure during space missions," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, Nov 2012, pp. 331–340.

[15] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *2013 13th International Conference on Quality Software*, July 2013, pp. 200–203.

[16] O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219 – 228, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404806001520

[17] H. Venter, J. Eloff, and Y. Li, "Standardising vulnerability categories," *Computers & Security*, vol. 27, no. 3–4, pp. 71 – 83, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404808000096

[18] Y. Younan, "25 years of vulnerabilities: 1988-2012," *Sourcefire Vulnerability Research Team*, 2013.

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

[20] H. Packard, "Fortify static code analyser," 2015, [online] http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

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

[22] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813604

[23] D. A. Wheeler, "Flawfinder," 2016, [online] http://www.dwheeler.com/flawfinder/.

[24] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 52–61.

[25] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, March 2011, pp. 249–258.

[26] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: ACM, 2008, pp. 23:304–23:318. [Online]. Available: http://doi.acm.org/10.1145/1463788.1463819

[27] I. Chawla and S. K. Singh, "An automated approach for bug categorization using fuzzy logic," in *Proceedings of the 8th India Software Engineering Conference*, ser. ISEC '15. New York, NY, USA: ACM, 2015, pp. 90–99. [Online]. Available: http://doi.acm.org/10.1145/2723742.2723751

[28] M. M. Ahmed, A. R. M. Hedar, and H. M. Ibrahim, "Predicting bug category based on analysis of software repositories."

[29] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference*, ser. ISEC '12. New York, NY, USA: ACM, 2012, pp. 125–130. [Online]. Available: http://doi.acm.org/10.1145/2134254.2134276

[30] L. Layman, A. P. Nikora, J. Meek, and T. Menzies, "Topic modeling of nasa space system problem reports: Research in practice," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901760

[31] T. F. of Incident Response and S. T. (FIRST), "Common vulnerability scoring system (cvss)," 2015, [online] https://www.first.org/cvss.

[32] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *2012 5th International Conference on Human System Interactions*, June 2012, pp. 89–96.

[33] T. M. Corporation, "Common weakness enumeration (cwe)," https://cwe.mitre.org/, 2015, online; accessed 20 August 2016.

[34] ——, "Common vulnerabilities and exposures (cve)," https://cve.mitre.org/, 2016, online; accessed 24 August 2016.

[35] ——, "Cwe-2000: Comprehensive cwe dictionary," 2015, [online] https://cwe.mitre.org/data/slices/2000.html.

[36] ——, "Cwe-1000: Research concepts," 2015, [online] https://cwe.mitre.org/data/graphs/1000.html.

[37] N. Mansourov, "Software fault patterns: Towards formal compliance points for cwe," 2011, [online] https://buildsecurityin.us-cert.gov/sites/default/files/Mansourov-SWFaultPatterns.pdf.

[38] T. M. Corporation, "Cwe-888: Software fault pattern (sfp) clusters," 2015, [online] https://cwe.mitre.org/data/graphs/888.html.

[39] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security Privacy*, vol. 3, no. 6, pp. 81–84, Nov 2005.

[40] T. M. Corporation, "Cwe-700: Seven pernicious kingdoms," 2015, [online] https://cwe.mitre.org/data/definitions/700.html.

[41] ——, "Cwe-699: Development concepts," 2015, [online] https://cwe.mitre.org/data/graphs/699.html.

[42] N. V. Database, "Cwe cross section mapped into by nvd," 2016, [online] https://nvd.nist.gov/cwe.cfm.

[43] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

[44] N. Project, "Natural language tooklkit," 2016, [online] http://www.nltk.org/.

[45] P. S. Foundataion, "Python," 2016, [online] https://www.python.org/.

[46] C. Donalek, "Supervised and unsupervised learning," 2011, [online] http://www.astro.caltech.edu/ george/aybi199/Donalek_classif1.pdf.

[47] V. N. Vapnik and V. Vapnik, *Statistical learning theory*. Wiley New York, 1998, vol. 1.

[48] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.

[49] M. L. G. at the University of Waikato, "Weka," 2015, [online] http://www.cs.waikato.ac.nz/ml/weka/.

[50] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Citeseer, 1998, pp. 41–48.

[51] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2, pp. 131–163, 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1007465528199

[52] L. Breiman and A. Cutler, "Random forests," 2015, [online] https://www.stat.berkeley.edu/ breiman/RandomForests/cc_home.htm.

[53] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[54] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[55] S. L. Developers, "Cosine similarity python implementation," 2016, [online] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html.

[56] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427 – 437, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306457309000259

[57] L. Manevitz and M. Yousef, "One-class document classification via neural networks," *Neurocomputing*, vol. 70, pp. 1466 – 1481, 2007, advances in Computational Intelligence and Learning 14th European Symposium on Artificial Neural Networks 200614th European Symposium on Artificial Neural Networks 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092523120600261X

[58] A. Dasgupta, P. Drineas, B. Harb, V. Josifovski, and M. W. Mahoney, "Feature selection methods for text classification," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 230–239. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281220

[59] S. Rose, D. Engel, N. Cramer, and W. Cowley, "Automatic keyword extraction from individual documents," *Text Mining*, pp. 1–20, 2010.

[60] M. Timonen, "Categorization of very short documents." in *KDIR*, 2012, pp. 5–16.

[61] M. Timonen, T. Toivanen, Y. Teng, C. Chen, and L. He, "Informativeness-based keyword extraction from short documents." in *KDIR*, 2012, pp. 411–421.

[62] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems*, 2015, pp. 649–657.

[63] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[64] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, March 2011, pp. 249–258.

[65] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10.

[66] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, Sept 2006, pp. 127–134.

[67] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: A case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985457

[68] M. Hafiz, P. Adamczyk, and R. E. Johnson, "Organizing security patterns," *IEEE Software*, vol. 24, no. 4, p. 52, Jul 2007. [Online]. Available: http://search.proquest.com/docview/215838715?accountid=2837

[69] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

Appendix A

CWE-888 Overview

Table A.1: CWE-888 Overview

CWE-888: Software Fault Pattern Overview						
Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
Risky Values			31			
	Glitch in Computation	31		partial	27	SFP1
Unused Entities			3			
	Unused Entities	3		yes	3	SFP2
API			28			
	Use of an Improper API	28		partial	20	SFP3
Exception Management			27			
	Unchecked Status Condition	17		partial	13	SFP4
	Ambiguous Exception Type	2		yes	2	SFP5

Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
	Incorrect Exception Behavior	8		partial	3	SFP6
Memory Access			20			
	Faulty Pointer Use	3		yes	3	SFP7
	Faulty Buffer Access	11		yes	11	SFP8
	Faulty String Expansion	2		yes	2	SFP9
	Incorrect Buffer Length Computation	3		partial	2	SFP10
	Improper NULL termination	1		singular	1	SFP11
Memory Management			5			
	Faulty Memory Release	5		yes	5	SFP12
Resource Management			17			
	Unrestricted Consumption	4		partial	3	SFP13
	Failure to Release Resource	7		yes	7	SFP14
	Faulty Resource Use	2		yes	2	SFP15
	Life Cycle	4		no	0	-
Path Resolution			51			
	Path Traversal	43		partial	38	SFP16
	Failed chroot Jail	1		singular	1	SFP17

Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
	Link in Resource Name Resolution	7		partial	4	SFP18
Synchronization			22			
	Missing Lock	13		partial	10	SFP19
	Race Condition Window	5		partial	4	SFP20
	Multiple Locks/Unlocks	3		yes	3	SFP21
	Unrestricted Lock	1		singular	1	SFP22
Information Leak			96			
	Exposed Data	76		partial	38	SFP23
	State Disclosure	7		no	0	-
	Exposure Through Temporary File	3		no	0	-
	Other Exposures	7		no	0	-
	Insecure Session Management	3		no	0	-
Tainted Input			138			
	Tainted Input to Command	87		partial	68	SFP24
	Tainted Input to Variable	8		yes	8	SFP25
	Composite Tainted Input	0		no	0	SFP26
	Faulty Input Transformation	15		no	0	-

Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
	Incorrect Input Handling	17		no	0	-
	Tainted Input to Environment	11		partial	3	SFP27
Entry Points			11			
	Unexpected Access Points	11		yes	11	SFP28
Authentication			43			
	Authentication Bypass	10		no	0	-
	Faulty Endpoint Authentication	11		partial	6	SFP29
	Missing Endpoint Authentication	2		yes	2	SFP30
	Digital Certificate	6		no	0	-
	Missing Authentication	2		yes	2	SFP31
	Insecure Authentication Policy	6		no	0	-
	Multiple Binds to the Same Port	1		singular	1	SFP32
	Hardcoded Sensitive Data	4		partial	2	SFP33
	Unrestricted Authentication	1		singular	1	SFP34
Access Control			16			
	Insecure Resource Access	4		partial	2	SFP35

Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
	Insecure Resource Permissions	7		no	0	-
	Access Management	5		no	0	-
Privilege			12			
	Privilege	12		partial	1	SFP36
Channel			13			
	Channel Attack	8		no	0	-
	Protocol Error	5		no	0	-
Cryptography			13			
	Broken Cryptography	5		no	0	-
	Weak Cryptography	8		no	0	-
Malware			11			
	Malicious Code	8		no	0	-
	Covert Channel	3		no	0	-
Predictability			15			
	Predictability	15		no	0	-
UI			14			
	Feature	7		no	0	-
	Information Loss	4		no	0	-
	Security	3		no	0	-
Other			46			
	Architecture	11		no	0	-

Primary	Secondary	# of CWEs	Primary CWE Totals	Pattern & Condition Available?	Discernible CWEs	SFP #
	Design	29		no	0	-
	Implementation	5		no	0	-
	Compiler	1		no	0	-
			632		310	36

Appendix B

Field Descriptions of Analyzed ITS's

Table B.1: IV&V Issue Tracking System Field Descriptions

IV&V Issue Tracking System Field Descriptions		
Column Title	Description	Is Field Used?
Issue ID	An ID is assigned to each issue and recorded in this field	Yes
Project	Contains the project name that the issue is from	Yes
State	The current state of the issue (i.e. Closed, Submitted, Withdrawn, etc)	Yes
Subject	The subject or title of the issue	Yes
Attachments	N/A	No
Capability	General Grouping of Functionality - A capability is made up of several subsystems which is made up of several software components	Yes
Comments	Updates about the progression and search for solution of the issue	Yes
Count	N/A	Yes
Data Restrictions	N/A	Yes
Defer Date	N/A	No
Defer Issue	N/A	No

Column Title	Description	Is Field Used?
Defer Notify Recipients	N/A	No
Description	The full description of the issue	Yes
Impact	The projected or observed impact of the issue on the system	Yes
Issue Category	The category the issue falls into (i.e. Code, Design, etc.)	Yes
Issue Type	The type of issue, more specific than Issue Category (i.e. Incomplete Design, Incorrect Code, etc.)	Yes
Severity	How sever the issue is	Yes
Method	The analysis method used to detect the issue	Yes
Originator	N/A	Yes
Phase Found	The project phase in which the issue was found	Yes
Phase Introduced	The project phase in which the issue was introduced	Yes
Phase Resolved	The project phase in which the issue was resolved	No
Recommended Actions	The action that the analyst recommends taking in response to the issue	Yes
References	Any material that can be reference to support the issue	Yes
Related Issues	Any issues highly related to the current issue	Yes
Resolution Chronology	A history of the solution of the issue	Yes
Technical Framework Level 1	N/A	Yes
Technical Framework Level 2	N/A	Yes
Technical Framework Level 3	N/A	Yes
Workaround	If issue cannot be solved, what was put in place to account for it	Yes
Defect	The defect of the issue (i.e. Software Behaviors, Requirements Documentation, etc)	Yes

Column Title	Description	Is Field Used?
Defect Category	The category of the defect (i.e. Design, Requirements, etc.)	Yes
Analysis Method	The method used to review the issue	Yes
Element	The element that the issue originates from, similar to the "Subsystem" in the Developer Issue Tracking System - A subsystem is made up of several software components	Yes
Date Submitted to POC	N/A	Yes
Reqt Number	N/A	No
Developer ITA	N/A	Yes
Verification Procedure Review	The procedure used to verify the fix of the issue	Yes
Created By	The analyst or developer that entered the issue into the issue tracking system	Yes
Created Date	The date the issue was entered into the issue tracking system	Yes
Updated By	The last analyst or developer to update the issue	Yes
Updated Date	The last data the issue was updated	Yes

Table B.2: Developer Issue Tracking System Field Descriptions

Developer Issue Tracking System Field Descriptions		
Column Title	Description	Is Field Used?
Issue ID	An ID is assigned to each issue and recorded in this field	Yes
DCR Product	The product the DCR relates to	Yes
Type	The project and product of the DCR	Yes
DCR Solution	The actions taken to resolve the DCR	Yes
DCR Severity	The criticality of the DCR	Yes
DCR Subtype	A general category of the problem under concern	Yes

Column Title	Description	Is Field Used?
DCR/Test Description	The description of the DCR	Yes
DCR Subsystem	The part of the system the DCR originates from, similar to "Element" from the IV&V Issue Tracking System - A subsystem is made up of several software components	Yes
DCR Type	The type of DCR (i.e. Defect, Change Request, etc.)	
DCR Title	The title or subject of the DCR	Yes
DCR Priority	How urgent fixing the DCR is	Yes
DCR Application	The application the DCR originates from	Yes
DCR Closure Notes	Points of interest detailing the solution of the DCR or deviations from normal routine	Yes
State	What lifespan stage the DCR is in	Yes
DCR Date Closed With Defect	N/A	Yes
DCR Date In Test	The date the DCR is ready for testing	Yes
Backward Relationships	Any previos DCR's that the current is related to	Yes
DCR Test Procs Used to Verify	The procedures used to verify the DCR	Yes
DCR Date On Hold	If the DCR was put on hold, the data of which this took place	Yes
DCR Date Test Completed	The date at which the testing on the DCR was completed	Yes
DCR Date Ready For Test	The date the DCR is ready to be tested	Yes
DCR Affects FSRL	N/A	Yes
Attachments	Any attachments that assist with the description, testing, or resolution of the DCR	Yes
DCR Date Closed	The date the DCR was closed	Yes

Column Title	Description	Is Field Used?
Implements	Any other DCR solutions that the DCR under concern implements	Yes
DCR Build Target	N/A	Yes
DCR Test Assigned Tester	The developer assigned to test the DCR	Yes
DCR Test Log Init Files Folder	N/A	Yes
Signature Comment	N/A	No
DCR IRB Comments	N/A	Yes
DCR Test Outcome	Initial test results	Yes
DCR Test Tester Comments	Comments left by the testing developer	Yes
DCR Date In Work	Date when work starts on the DCR	Yes
DCR Date Work Completed	Date the work is finished on the DCR	Yes
DCR Phase Found	The development phase of which the DCR was found	Yes
DCR IRB Comments History	N/A	No
DCR Workflow	N/A	Yes
Forward Relationships	The related DCR's created after the one under concern	Yes
DCR Date Assigned	The date the DCR is assigned to a dveloper	Yes
DCR Test Log Init Files	N/A	Yes
DCR Document Type	The type of DCR document (i.e. requirements, algorithms, etc)	Yes
Links to Tests from DCR	The tests relevant to the DCR	Yes
DCR Additional Products Affected	Products other than the one the DCR originated from that are effected	Yes
Modified Date	The last date the DCR was modified	Yes

Column Title	Description	Is Field Used?
DCR Date Ready For Closure	The date the DCR is marked as ready to close	Yes
Signed By	N/A	No
Modified By	The developers to modify the DCR	Yes
DCR Test Verification	How was the DCR verified	Yes
DCR Date Build Integration	The date the DCR was integrated	Yes
DCR/Test Leads Comments	Project lead comments	Yes
DCR Test Log Folder Verify	N/A	Yes
DCR Test Log Files Verify	N/A	Yes
Is Related To	Other DCR's the one under concern is related to	Yes
DCR Assigned To	The Developer the DCR was assigned to	Yes
Created By	The creater of the DCR	Yes
DCR Build Found	The build in which the DCR was found	Yes
DCR Test Procs Init Used	N/A	Yes
Assumed Issue Category	N/A	No

Empirical Analysis and Automated Classification of Security Bug Reports

Jacob P. Tyo

Thesis submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Lane Department of Computer Science and Electrical Engineering

APPROVAL OF THE EXAMINING COMMITTEE

Roy S. Nutter, Ph.D.

Matthew C. Valenti, Ph.D.

Katerina Goseva-Popstojanova, Ph.D., Chair

Date