DETECTION OF GENERALIZABLE CLONE SECURITY CODING BUGS USING GRAPHS AND LEARNING ALGORITHMS

Quentin Reuben Mayo, B.S., M.S.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2018

APPROVED:

Renee Bryce, Major Professor
Ram Dantu, Co-Major Professor
Suliman Hawamdeh, Committee Member
Dan Kim, Committee Member
Mark Thompson, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Yan Huang, Interim Dean of the College of Engineering
Victor Prybutok, Dean of the Toulouse Graduate School

Mayo, Quentin Reuben. *Detection of Generalizable Clone Security Coding Bugs Using Graphs and Learning Algorithms*. Doctor of Philosophy (Computer Science and Engineering), December 2018, 120 pp., 17 tables, 59 figures, 53 numbered references.

This research methodology isolates coding properties and identifies the probability of security vulnerabilities using machine learning and historical data. Several approaches characterize the effectiveness of detecting security-related bugs that manifest as vulnerabilities, but none utilize vulnerability patch information. The main contribution of this research is a framework to analyze LLVM Intermediate Representation Code and merging core source code representations using source code properties. This research is beneficial because it allows source programs to be transformed into a graphical form and users can extract specific code properties related to vulnerable functions. The result is an improved approach to detect, identify, and track software system vulnerabilities based on a performance evaluation. The methodology uses historical function level vulnerability information, unique feature extraction techniques, a novel code property graph, and learning algorithms to minimize the amount of end user domain knowledge necessary to detect vulnerabilities in applications. The analysis shows approximately 99% precision and recall to detect known vulnerabilities in the National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project. Furthermore, 72% percent of the historical vulnerabilities in the OpenSSL testing environment were detected using a linear support vector classifier (SVC) model.

Copyright 2018

by

Quentin Reuben Mayo

ACKNOWLEDGMENTS

I would like to express thanks to the many people who helped me through my academic journey. It takes a village to raise a child; it also takes a community working together for an individual to obtain their Ph.D. I would like to thank my committee members, Dr. Hawamdeh, Dr. Kim, and Dr. Thompson for their guidance and support. Special thanks to my advisor, Dr. Renee Bryce, for allowing me the opportunity to pursue a Ph.D. and for seeing my potential. Your guidance, patience, and mentorship during this process contributed to my success. I would also like to thank Dr. Dantu for funding, guidance, and mentorship. You always challenged me to do better and, for that, I say thank you.

I am particularly grateful to my wife, Jenny for supporting and encouraging me in every way, including proofreading my papers. To my sons, Andrew and Reuben: thank you for being a source of inspiration.

I want to give special thanks to my parents, who taught me that consistency and persistence were key to my success. I am grateful to my late brother, Dr. Jessie Benjamin Mayo, Jr. He was indeed a pioneer who showed me that obtaining a Ph.D. was possible. Thanks to my brother, Dr. David Mayo, for calling me every day just to motivate me to keep writing.

Finally, I am grateful to the Software Testing Lab at Discovery Park for being my extended family. I could always brainstorm with them on my research-related questions.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 MOTIVATION AND BACKGROUND 6

2.1. Objective 7

2.2. Contributions 7

2.3. Threats to Validity 8

2.4. Bugs and Security 10

2.5. Relationship between Security and General Bugs 13

2.6. Detour in Compiler Theory 13

2.7. Source Representation Analysis 19

2.8. Limitations of Source Representation Analysis and Other Research 19

2.8.1. Program Representations 20

2.8.2. Null Pointer 27

2.8.3. Resource Drains 28

2.8.4. Integer Overflow 29

2.8.5. Number Handling 29

2.8.6. Memory Corruption 30

2.8.7. Concurrency Handling 31

2.8.8. Injection 31

2.8.9. Other Vulnerabilities 32

2.9. Literature Review 32

2.10. Performance Metrics 36

CHAPTER 3 DESIGN AND PROCEDURE 38
3.1. Hybrid TF-IDF for Vulnerability Detection 38
3.2. NIST SAMATE 39
3.3. Core Graph Representation 43
3.3.1. Program Structure and Composition 44
3.4. System Configuration 48
3.5. Current Version of OpenSSL and Commit History 48
3.6. Experiment Setup 56
3.7. Learning Models 60
3.8. Extracting Meaning from the Core Graph 61
3.9. Addressing the Core Graphs Features and Classes Problem 64
3.10. Other Approaches: Bug Prediction 65

CHAPTER 4 RESULTS 68
4.1. Analysis of the Data 68
4.2. NIST SAMATE 69
4.2.1. DT OpenSSL NIST: Simple Classification Using Edges with OpN 69
4.2.2. Random Forest Classifier OpenSSL NIST: Simple Classification Using OpEdges with Op 71
4.2.3. SGD Classifier OpenSSL NIST: Simple Classification Using Edge with OpN 73
4.2.4. DT Classifier NIST: Simple Classification Using Edge with Op1 74
4.2.5. Random Forest Classifier NIST: Simple Classification Using OpEdge with Op1 75
4.2.6. SGD Classifier NIST: Simple Classification Using Edge with Op1 76
4.3. OpenSSL 78
4.3.1. DT Classifier: OpenSSL 79
4.3.2. Random Forest Classifier: OpenSSL 84
4.3.3. Linear SVC Classifier: OpenSSL 88

4.3.4. SGD Classifier: OpenSSL 90
4.4. OpenSSL: A Comparison of Different Bug Prediction Learning Approaches 95
4.4.1. OpenSSL: Halstead Core Features 95
4.4.2. OpenSSL: Halstead Features 96
4.4.3. OpenSSL: Nguyen 97
4.4.4. OpenSSL: Wang 97
4.5. Summary Performance Reports 98
CHAPTER 5 DISCUSSION 102
5.1. Samate Dataset 102
5.1.1. Vul Misclassification Problem 102
5.2. The Pre-Post Patch Problem 105
5.3. Current Version of OpenSSL 106
5.4. Pre-Patch Problem 107
5.5. Patch Limitations 108
CHAPTER 6 CONCLUSION 110
CHAPTER 7 FUTURE WORK 112
REFERENCES 115

LIST OF TABLES

Page

Table 3.1. SAMATE Applications 45

Table 3.2. OpenSSL Version Information 49

Table 3.3. Graph Properties 59

Table 3.4. Feature Extraction Techniques 62

Table 4.1. DT OpenSSL NIST Report: Simple Classification Using Edges with OpN 71

Table 4.2. RandomForest Classifier OpenSSL NIST Report: Simple Classification Using OpEdges with Op 72

Table 4.3. SGD Classifier OpenSSL NIST Report: Simple Classification Using Edge with OpN 73

Table 4.4. Random Forest Classifier NIST Report: Simple Classification Using OpEdge with Op1 76

Table 4.5. SGD Classifier NIST Reports: Simple Classification Using Edge With Op1 78

Table 4.6. DT Classifier Report: OpenSSL 81

Table 4.7. Random Forest Classifier Report: OpenSSL 86

Table 4.8. Linear SVC Classifier Report: OpenSSL 90

Table 4.9. SGD Classifier Report: OpenSSL 94

Table 4.10. NIST Vulnerability Classification Prediction Summary 99

Table 4.11. SARD F1 Score Vulnerability Classification Summary 100

Table 4.12. OpenSSL Vulnerability Classification Summary Report 101

Table 5.1. Report: NIST Dataset CWE Mapping using DT 105

LIST OF FIGURES

Page

Figure 1.1. Requirement and Implementation Coding Issues [6] 1

Figure 1.2. Software Analysis Scope (Execution Time vs Analysis Type) [6] 2

Figure 2.1. Translation of an assignment statement [1] 15

Figure 2.2. LLVM IR Pass Flow [31] 16

Figure 2.3. Source Code to Machine Code [38] 17

Figure 2.4. The Software Similarity Problem [5] 20

Figure 2.5. Vulnerabilities breakdown [18] 26

Figure 2.6. Number Handling [12] 29

Figure 2.7. History of software defect prediction studies [37] 34

Figure 3.1. Samate Programs [11] 40

Figure 3.2. Samate Number of Complexity [11] 41

Figure 3.3. Weakness Classes [11] 42

Figure 3.4. BP SM OSSL TS Count 46

Figure 3.5. PP SM OSSL TS Features 47

Figure 3.6. PG SM OSSL TS Features 47

Figure 3.7. Attacks Through Time 50

Figure 3.8. OpenSSL Function Calls in Diff Changes 51

Figure 3.9. Line Modifications in Diff Changes 52

Figure 3.10. OpenSSL CVE: Authors vs. Reviewers vs. Committers 52

Figure 3.11. CVE 2014 3512(Old): Pre-Source Code Change 53

Figure 3.12. CVE 2014 3512(New): Post-Source Code Change 53

Figure 3.13. OpenSSL: Source Code Diff Features 54

Figure 3.14. OpenSSL: Vulnerabilities throughout the years 55

Figure 3.15. OpenSSL: Vulnerabilities Applications 55

Figure 3.16. Framework 57

Figure 3.17. Workflow 58

Figure 4.1. DT OpenSSL NIST Confusion Matrix: Simple Classification Using Edges with OpN 70
Figure 4.2. RandomForest Classifier OpenSSL NIST Confusion Matrix: Simple Classification Using OpEdges with Op 71
Figure 4.3. SGD Classifier OpenSSL NIST Confusion Matrix: Simple Classification Using Edge with OpN 74
Figure 4.4. DT Classifier NIST Confusion Matrix: Simple Classification Using Edge with Op1 75
Figure 4.5. SGD Classifier NIST Confusion Matrix: Simple Classification Using Edge with Op1 77
Figure 4.6. SGD Classifier NIST ROC: Simple Classification Using Edge with Op1 78
Figure 4.7. DT Classifier Normalize Confusion Matrix : OpenSSL 79
Figure 4.8. DT Classifier Confusion Matrix: OpenSSL 80
Figure 4.9. DT Classifier Cumulative Gain Curve: OpenSSL 81
Figure 4.10. DT Classifier Life Curve: OpenSSL 82
Figure 4.11. DT Classifier Precision Recall Chart: OpenSSL 83
Figure 4.12. DT Classifier ROC Curve: OpenSSL 83
Figure 4.13. Random Forest Classifier Normalized Confusion Matrix: OpenSSL 84
Figure 4.14. Random Forest Classifier Confusion Matrix: OpenSSL 85
Figure 4.15. Random Forest Classifier Cumulative Gains: OpenSSL 85
Figure 4.16. Random Forest Classifier KS: OpenSSL 87
Figure 4.17. Random Forest Classifier Lift Curve: OpenSSL 87
Figure 4.18. Random Forest Classifier Precision Recall Curve: OpenSSL 88
Figure 4.19. Random Forest Classifier ROC Curve: OpenSSL 89
Figure 4.20. Linear SVC Classifier Normalize Confusion Matrix: OpenSSL 89
Figure 4.21. Linear SVC Classifier Confusion Matrix: OpenSSL 90
Figure 4.22. SGD Classifier Normalize Confusion Matrix: OpenSSL 91
Figure 4.23. SGD Classifier Confusion Matrix: OpenSSL 92

Figure 4.24. SGD Classifier Cumulative Gains: OpenSSL 92
Figure 4.25. SGD Classifier Lift Curve: OpenSSL 93
Figure 4.26. SGD Classifier Precision Recall Curve: OpenSSL 93
Figure 4.27. SGD Classifier ROC Curve: OpenSSL 94
Figure 4.28. Halstead Core Features Linear SVC Classifier Normalize Confusion Matrix: OpenSSL 95
Figure 4.29. Halstead Features Linear SVC Classifier Normalize Confusion Matrix: OpenSSL 96
Figure 4.30. Nguyen Features with Linear SVC Classifier Normalize Confusion Matrix: OpenSSL 97
Figure 4.31. WangFeatures with Linear SVC Classifier Normalize Confusion Matrix: OpenSSL 98
Figure 5.1. Normalized Confusion Matrix: NIST Dataset CWE Mapping using DT 103
Figure 5.2. Confusion Matrix: NIST Dataset CWE Mapping using DT 104

CHAPTER 1

INTRODUCTION

System bugs such as faults/errors and security vulnerabilities are significant concerns for software applications. Any coding operation that causes the system to behave unusually is defined as a bug. When addressing buggy software applications (defined as those in which there are numerous coding errors within a particular software process), some side-effects include user interface (UI) issues or inconsistent output. Many software application bugs cause invalid output to the user, crashes of the application, and performance issues [8]. Additionally, security bugs are also system vulnerabilities; these bugs increase the chance for the application and system to be exploited.

To prevent bugs and reduce vulnerabilities, we need software security. Software security is the practice and art of building security into the code or development process to reduce vulnerabilities. Most software projects create artifacts, which include the source code, but some additional artifacts include the creation of bugs. Often, security is considered an afterthought during application development and many applications are not resilient to attacks [6]. Security bugs are also common when software developers use vulnerable APIs such as strcpy in the C programming language.

A widespread practice is to reduce the number of coding mistakes and errors by improving the code with quality assurance measures. However, writing code leads to writing/scripting mistakes that manifest as bugs. It is also difficult to improve security by only

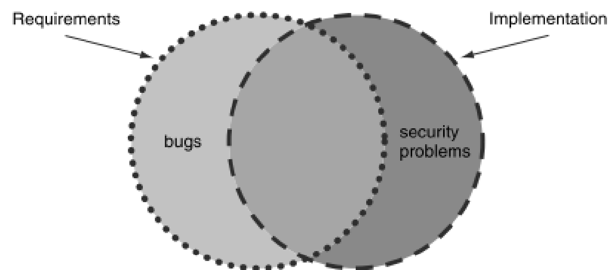

Figure 1.1. Requirement and Implementation Coding Issues [6]

improving quality assurance because software assurance testing focuses on testing functionality and often does not include security requirements. If security requirements are also poorly communicated or inadequate, it may be difficult for a programmer to understand those security requirements, given their complexity. This lack of understanding leads to increased security risk. Figure 1.1 illustrates the relationship between requirements and implementation [6]. Software system requirements overlap implementation, which encompasses bugs and security problems. The combination of bugs and security problems cause major issues in an application. In general, standard system requirements do not cover many security requirements.

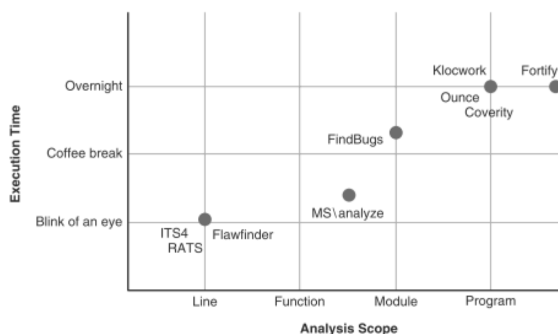

FIGURE 1.2. Software Analysis Scope (Execution Time vs Analysis Type) [6]

To improve applications when dealing with security issues and bugs, the advent of code reviews and commercial tools help in addressing software system security issues. Figure 1.2 shows various tools and features for analysis [6]. Different tools have different scopes and execution times for analyzing applications. Theses analysis scopes could help companies understand applications better compared to manual analysis approaches. However, companies still employ expertise in both security- and software-related fields to perform code reviews. Depending on the size and complexity of software, these code reviews take days or months to complete and can still have system issues. These reviews include functional and nonfunctional testing. While functional and nonfunctional testing on an application examines the requirements, these tests often do not focus on the security of applications. In many cases, these tests overlook security problems because security problems might not

violate the requirements.

For companies that do use tools, they have an assortment of issues. In general, industry professionals focus on commercial applications for bug and security vulnerability detection in code [27, 28], while academics and researchers focus more on open source tools. Companies spend effort on commercially-developed tooling due to the lack of usability and single functionality of many open source tools. However, there drawbacks for commercial and open source tools. Commercial applications are expensive, which deters many academics from using them. Many open source tools require a high-level of instrumentation for code analysis. Commercial applications aim for generalization in support languages, while open source tools focus on specific security bugs. Business tools such as AppScan can find numerous issues in the source code, while many open source tools can only identify a few problems in a given program[1]. Both sets of tools are prone to false positives (sections of code that have been misidentified as vulnerabilities by tools).

Many of the commercial tools used for analyses of programs are prohibitively expensive for the individual consumer. In addition, the current tooling do not detect bugs effectively, thereby increasing the cost to find and address these bugs; companies, such as Google, offer bug bounties upwards of thousands of dollars to locate bugs only in one application. The reward amount can increase for mission-critical or government applications as a small bug can result in life threatening situations. Due to the cost of commercial tools for identifying bugs, many companies outsource their bug and vulnerability detection. However, this process is also expensive. External analyses of applications with commercial tools can cost twice as much as the actual product.

Commercial and open source tools also struggle with false positives because software-related security bugs are a severe problem at the design and implementation level and can be hard to detect. In critical systems, flaws can lead to security attacks [27, 28]. While software testing offers a means of eliminating software bugs, this can be expensive (in terms of money, time, and resources). The cost of software bugs is directly related to the time

[1] http://www-03.ibm.com/software/products/en/apps

programmers spend fixing bugs [22]. Maintenance, including handling security issues, can make up over 70% of the total life-cycle cost for a given program, and the number of bug reports can exceed the technical staff available to work on such problems. Many bug reports generated through routine maintenance include duplicate bugs; for example, studies have revealed that as many as 36% of bug reports were duplicates or invalids, which can further increase costs [22]. In 2005, software maintenance cost companies $86 billion dollars [17]. This number has increased over the years. Reporting bugs is costly for everyone involved in the development process. A report by Hay et al. [20] stated that, according to the Department of Commerce, activities focused on improving the quality of bugs accounted for 50% of the budget for the average software company and cost companies $59.5 billion annually.

There are many security and common bug types reported for various applications [29, 30]. These bugs can manifest at the coding level, such as a null pointer caused by a unique edge condition in an application. They can also stem from communication issues between multiple applications. Further, vulnerabilities such as cross-site scripting in the application and interconnection between complex applications are issues for companies. In order to address bug reports, developers must diagnose the cause, produce a patch that could fix the bugs, and commit the patch to a repository. This process involves bug triaging, which may require particular expertise. An expert would handle patching a DOS attack differently from resolving a source code overflow, and specific knowledge would be required to understand both coding issues. Experts fixing bugs in application require modifying foundational sections of the code base. Therefore, information collection and bug triaging can be expensive and time-consuming processes that can take days or weeks.

Categorizing bugs is also a unique challenge for experts in software security and bug analysis when dealing with many bug types. For example, all security bugs are bugs, but not all bugs are security bugs. Although it is desirable to generate patches for all bug reports in a generalized format, realistically, that is extremely difficult [27]. Many bug reports are invalid, incomplete, or unrepresentative, which means it is challenging to build automatic

fixes. Using only security bug fix reports is not a viable option for understanding the security risk involved with a system. Some solutions require a full redesign of the algorithm and additional features. Classifying bugs is challenging because, with the compilation process levels, bugs have different meanings. Bugs would include finding null pointers and be off by one error [2, 23, 24]. Commercial software applications identify errors such as cross-site scripting (XSS) and authentication issues.

Many security tools can handle inter-language problems. Past research on tools has focused on theories for finding issue in multiple languages [16]. The advantage of the theoretical approaches is that many of the ideas and algorithms can extend to numerous languages and applications. Open source tools use theoretically-based algorithms and have several limitations when finding issues in applications. These theoretical foundation limitations also occur in commercial applications. A well-known example of this issue is defects characterized as generic and context-specific [6]. Generic flaws, such as buffer overflows and exception handling, are common to all programs. Depending on implementation, there may be some difficulty associated with determining the cause of a buffer overflow. Context-specific defects are created in formal requirement documents. Just as it can be challenging to identify a generic defect, finding context-specific errors is extremely difficult when using a detection algorithm. However, during the analysis process, it is important to consider both defects visible to debugging algorithms and other software issues that are visible in the design phase. Security bugs such as input validation, API issues, and error handling can be classified as either general or specific context-based bugs.

CHAPTER 2

MOTIVATION AND BACKGROUND

Addressing software bugs is important and it is currently necessary to utilize manual analysis to understand a program. Manual analysis is a process during which the coder searches for flaws. However, bugs are often found after an exploit has occurred. Manual analysis can be time-consuming; if the exploit is not detected, the vulnerability can take years to fix [13]. An approach for addressing such an issue is to use patch information, which provides details related to how a code has changed over time. This allows for the use of domain data that cannot be captured easily with only static analysis. Static analysis can help in terms of analyzing the code before compiling, but it is difficult to introduce static analysis tools during the latter phase of a software project. It is also necessary to implement software diagnosis tools into the build process rather than scanning code at the lexical level. This would allow for the detection of software clone coding, which has increased in the development process as consistent changes can cause some issues for developers. Furthermore, research by Li et al. has shown that semantic issues are the cause of many bugs [26]. Typos or copy-paste actions can be the source of various software bugs. The key is to avoid verbose solutions in order to mitigate both false positives and false negatives [26]. Thus, basing the automated static analysis on patching can reduce the overall quantity of false positives.

While research has shown that there is a benefit to using traditional static analysis, there are still problems with current static or manual analysis tool-chains (e.g., concurrency, memory, and semantic bugs), which indicates that further research is necessary in this area [41]. Currently, only a small amount of extant research has addressed utilizing patch information to detect potential software bugs [3, 39, 40, 47]. Also, some programming research has considered the minimization of plug-in tools to address failure side effects within software applications [34, 39]. However, extant research has not focused on fixing the cause of the issue in the system.

The motivation for this research is to address issues associated with preemptively

locating, identifying, and mitigating software bugs by leveraging patch information. To achieve the motivation of this work, I create techniques and approaches that complement others using learning techniques. One of the goals of this work is to find at-risk functions, which include risky functions and operations that may not be currently vulnerable. By using learning techniques and creating workflow, these functions or locations of risky methods can be noted and tested with an application security testing tool. Vulnerabilities and at-risk functions can also be reviewed by domain experts or developers who know how to mitigate the issue.

2.1. Objective

The objective of the present work is to create a code analysis framework to support a software build process. This will require code analysis that identifies the root cause of some vulnerabilities. LLVM has been shown to be a pivotal compiler infrastructure to address code analysis while maintaining the code properties at an intermediate level. The secondary objective of this research is to automate the process and evaluate code analysis techniques using learning algorithms. These objectives can be summarized as follows:

- Build a core graph representation of a program that captures relevant vulnerabilities.
- Develop novel feature extraction and learning models for predicting vulnerabilities or at-risk methods in applications using an intra-procedural analyses.
- Test against different datasets (NIST's SARD datasets and OpenSSL)
  - Analyze program code
  - Evaluate learning models

2.2. Contributions

The contributions of this research are a hybrid code property graph using LLVM that maintains source code metadata, a framework for testing vulnerabilities based on clone code detection, and an analysis of the approach against Software Assurance Metrics And Tool Evaluation(SAMATE) project testing dataset and OpenSSL. I executed vulnerabilities represented by LLVM and analyzed the effectiveness. The approach focuses on detecting vul-

nerabilities based on historical data. The overall contribution is defining practical methods for finding security-relevant coding issues. This study's contributions include

- Hybrid code property graph
- Framework for testing vulnerabilities based on clone code detection
- Test against different datasets (NIST's SARD datasets and OpenSSL)
- Vulnerability Analytics using the hybrid code property graph

2.3. Threats to Validity

Throughout the project, I discovered several threats to validity involving this study's methodology and domain challenges in security. One threat to validity was that I used only modules (files) with known vulnerabilities for analysis. The decision to focus on fewer modules was based on the assumption that, as the size of the graph increases, the time it takes for analysis would dramatically increase. Thus, to minimize the number of modules and reduce the runtime, I focused on files with known vulnerabilities. Additionally, this led to a focus on riskier modules that had known vulnerabilities over other modules without any known issues. Since I mapped vulnerable functions tested on issues seen during normal operation, the modules under investigation have functions related to code that runs during normal user operation. If this study focused on analyzing functions across a whole code base, the tested models would have required evaluation of a great deal of code that is not used during normal operation. This includes Quality Assurance(QA) testing code and specific testing code for testing SSL clients. When focusing only on vulnerable patch changes in the associated file on OpenSSL, the models target the directories involving apps, crypto, engines, and SSL in the main OpenSSL project. Many of the functions tested are key parts of the OpenSSL Library.

Limited configuration regarding builds constituted another threat to validity. I used a standard configuration when building OpenSSL on an Ubuntu workstation. This means that one could assume that the analysis is limited to only Ubuntu systems with similar configurations. However, the code only have slight variations depending on the platform and configuration used during compile time. For example, OpenSSL can be compiled on BSD.

While compiling on different systems causes different code changes, in most cases, the overall code structure is the same.

The next threat to validity involves zero knowledge vulnerabilities. Vulnerabilities have a time window involving the discovery, the actual exploit on the system, and when the patch is applied to the system. Usually, in the process of discovering a vulnerability, an exploit is created that leverages the vulnerability. The next phase is the development of the zero knowledge vulnerability, which is the vulnerability itself. Zero knowledge does not mean hackers do not know of the exploit or are not exploiting the application, but that the public does not yet know of the issue. The vulnerability window is the time it takes for an IT admin to fix the issue after becoming aware of the problem. While this dissertation provides a foundation for finding at-risk functions, this does not fix the issue. While this is a threat to validity, it is not a limitation because the responsibility of assessment, mitigation, and repair of any issue is up to specific users of any applications; the approach discussed in this study aims to help users find locations in the code that have a high risk of issues.

Another threat to validity of the study's methodology is the inability to find exact vulnerabilities. The approach developed in this dissertation does not allow for the identification of exact vulnerabilities because it focuses on instruction types to determine at-risk functions. By looking at function types and converting the data into dot notation that maintains information on the application, such as the locations in code, precision in terms of finding the actual vulnerability is lost. However, focusing only on the instruction types allows for generalization of the application so that data can be constructed as learning based problems.

The final issue when dealing with patch data stored in a version control repository is that a vulnerability patch can lead to new vulnerabilities. Patches for major vulnerabilities are released fast, and a given patch might not fix the vulnerabilities. When a zero-day is release and is publicly available for a major application or tool suite, a patch can be available in a few days, A hotkey, which is a quick solution that should mitigate an issue for an application, can be available publicly within hours. When patches or hotfixes are released

fast, these patches or hotfixes might not fix the original problem and instead could create new problems. Patches can create other vulnerabilities, which can cause new issues for development teams. I addressed this concern in several ways throughout this study. First, I focused on trusted, third-party organizations and data sources such as the National Institute of Standards and Technology (NIST), the National Vulnerability Database (NVD), and Common Weakness Enumeration(CWE) to verify and confirm vulnerabilities and patch data. These trusted sources include review of many stages involved with validating patching for major open-source projects. Large teams such as the OpenSSL development team review patches to understand whether the application patch fixes the issue or not. With critical applications, such as OpenSSL which is used to secure almost all applications and the Internet, patches and hotfixes are reviewed by a large amount of experts before being released to the public.

Even though bad patching is a concern, Li and Paxson [25] found that only 7% of patches failed to completely remedy a security hole. This means that 93% percent of all security patches do fix the issue. Patch information can be better than bug reports which can go unused by development teams. Weimer found that patch data is more relevant than bug reports because bug reports with relevant patch information are more likely to be fixed [49]. In this study, if a patch fixed an issue but led to another vulnerability, I translated the two patches into two different vulnerabilities. Each vulnerability and patch had code properties that could be extracted. This process of capturing information regarding multiple vulnerabilities is the same as extracting features for one issue.

2.4. Bugs and Security

Software bugs within code are different from security bugs; however, further research on this topic is necessary. There are several software bug types, including performance, user interface, and coding errors. Bug types are categorize further into many sub-categories. Performance bugs can affect an entire system and are often classified as “optimization bugs, security bugs, or as security performance bugs” [28, 53]. User interface bug issues are related to the software/human, interface, and system as these are the means for which attackers can

exploit user assumptions and behavior. Further, software coding errors are issues within programming that may include null pointers and off-by-one mistakes. Many performance bugs relate to the optimization of the system, or inefficiency, and can affect system security by over burdening and taxing the system. Coding an algorithm to more efficiently accommodate new or current tasking is a more desirable strategy, where any mistake in a hashing algorithm will generate a wrong hash that can potentially be exploitable. As the level of abstraction and complexity increases within a software code, finding a bug and understanding its effects becomes critical. Software programming complexity may require new tools and skills to be developed to address the evolving issue of bug identification. Increased software program complexity and the lack of expertise to address the vulnerability could contribute to an increased number of zero-day vulnerabilities.

Dowd et al. addressed a contrast between bugs and vulnerabilities [33]. Dowd et al. noted that "vulnerabilities are specific flaws or oversights in a piece of software that allows attackers to do something malicious and these are errors that create undesirable behaviors". Their work provides a foundation to connect security vulnerabilities as a specific bug type. However, this could lead to "application exploitation," which is the process of taking advantage of a software system that has a security vulnerability. While security policies may prevent such exploits, these security policies often go unused. For example, a well-known system vulnerability would be cross-site scripting (XSS) which can be located by accessing web pages. Given the amount of the requests a website sends between each page, there are many locations a malicious user can leverage to exploit a system. Thus, many XSS issues go unfixed.

Additionally, there are three main software language-related bug categories: logical, implementation, and coding errors [6]. Logical issues stem from errors that arise from the design implementation of a software application. For example, missing functional requirements at the design level can create security or general issues. If a programmer adds a button to UI that was not in the requirements, that implementation would be considered as a logical issue. The concern is not related to the software coding, but rather to the lack of

implementation or the logic in negating the original framework requirements. Implementation issues are defined by the software code rather than the logic. Specifically, if the logic is correct/consistent with respect to the requirements, but the system has issues, it can be defined as an implementation issue. For example, consider a program that calculates the mean of financial records. An implementation error would be a miscalculation that would result in an erroneous final result. The issue is likely not logical, because the programmer's logic is sound. Therefore, the issue falls into implementation error category. Finally, coding errors can be divided into two groups: run-time errors and compiler errors, which are strongly dependent on the software language. For example, GCC and G++ are likely to have no compiler warning messages using default configuration for Null pointer issues. Null pointers is commonly used to initialize a pointer when the object reference is unknown. When that pointer is referenced while Null, there can be significant security issues. Referencing a null pointer will retrieve information that may crash the system. The main difference between run-time errors and compiler issues is that compiler issues can be detected when compiling code, while run-time issues are usually difficult to show before computation execution.

Dowd et al. further classified vulnerable and common threats related to design and implementation [33]. They have an additional classification referred to as "operational bugs." These design vulnerabilities are also logic errors and are a result of an oversight in the software design. In this case, the developers would need to address critical issues such as vague or misunderstood requirements. Implementation vulnerabilities present a problem when a task is preformed incorrectly. This occurs during the implementation phase in the Software Development Life Cycle. Operational vulnerabilities are unique because they happen during the execution of an application. In many cases, these issues are caused by compatibility issues with the application and the software environment. There can be several other causes such as issues resulting from social engineering. These issues could include: input and data flow issues, trust relationship, misplaced trust, and the assumptions developers and business leaders have concerning the end user. It is also important to note that input and data flow issues account for the majority of problems with software applications.

2.5. Relationship between Security and General Bugs

While all security bugs are bugs, not all bugs are security bugs. Security bugs have a direct relationship to confidentiality, integrity, and availability (the CIA triad). Confidentiality is very similar to privacy in that the goal is to stop attacker from releasing sensitive information. For example, a bug can allow for an attacker to read an application's memory, including addresses that should not have been accessed. Personal information, such as API credentials or banking details changes, are saved at specific memory addresses, which affects the confidentiality of an application. Bugs can be caused by unsanitized input to a system, which presents as a buggy input field that does not respond properly. Integrity is closely related to the stability of an application. The application should maintain the accuracy of the data. However, if an attacker figures out how to do stack smashing, they can crash the application or change data to cause an application to send the wrong information to different processes. This will affect the output of the application. Stack smashing can also be an example of issues in availability. If an attack brings down the system, it has to have a direct effect on the application. All these issues originate from the bug that also has security implementations.

2.6. Detour in Compiler Theory

Compiler theory is a foundational topic for software development and code related vulnerabilities research [1]. Software focused literature is rooted in compiler theory, programming languages, or automata theory, and these topics have overlapping concepts. The key concept in compiler theory is computability and how a compiler functions. All languages, including compiler or scripting, go through phases of computation. In general, a compiler uses a source program and outputs targeted machine code to produce a typical compile language such as C/C++. With a target program, input is passed through a compiler to produce an output. A executing program may not use any input. With interpreters, the machine interprets a source program and input to generate an output. Python or JavaScript are script interpreters that aid the programming process. The key difference between the interpreter and compiler is that the compiler produces a target program proper for execution. Most

modern languages rarely employ pure interpreters. In general, there are more hybrid than pure interpreters. When Python compiles code to a .pyc file, it becomes a hybrid compiler. Hybrid compilers translate the source program into an intermediate program. The generated intermediate program feeds into a virtual machine which produces an output. These concepts are important because CLANG/LLVM is a hybrid compiler that creates LLVM-IR code for analysis. CLANG is a front-end compiler for languages such as C and C++. The original goal of Clang was to create a new, C-based language front-end. LLVM provides the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and producing an improved IR. There are several advantages to using an intermediate program, compared to using binaries or raw source code. The key benefit to using intermediate program operations is that it allows for the extraction of program representations without lexical issues for application security research.

When considering the static analyzers that address analysis at distinct levels of applications, most of the work is based on compiler theory concepts. Figure 2.1 shows the translation process for a compiler [1]. The foundation is the source program and source code. The source code depends on the standards and the requirements for the compiler, but it is possible to write a program without compiling it. Research by Mark et al. focused on creating a collection of documents for finding software vulnerabilities [33]. Much of the research mentioned in Mark et al. focus on the application without analyzing code properties [33] and focus on analyzing a program at a lexical level. However, it is necessary to focus on code properties and extracting information from the source code. Further, some static analysis tools focus on user-defined variables to infer information about the program which is a problem. During a raw code level analysis, analysis works on information provided at a higher levels of the compilation phases. For example, if there is a hash public key encryption program, but the application's function parameter takes a private key for a method call, and the parameter name of the function's first variable is public, there would be an issue. Static analysis at a source level would address this issue but not be able to address issues beyond a lexical level. A solution would be to name the public key data as a private key.

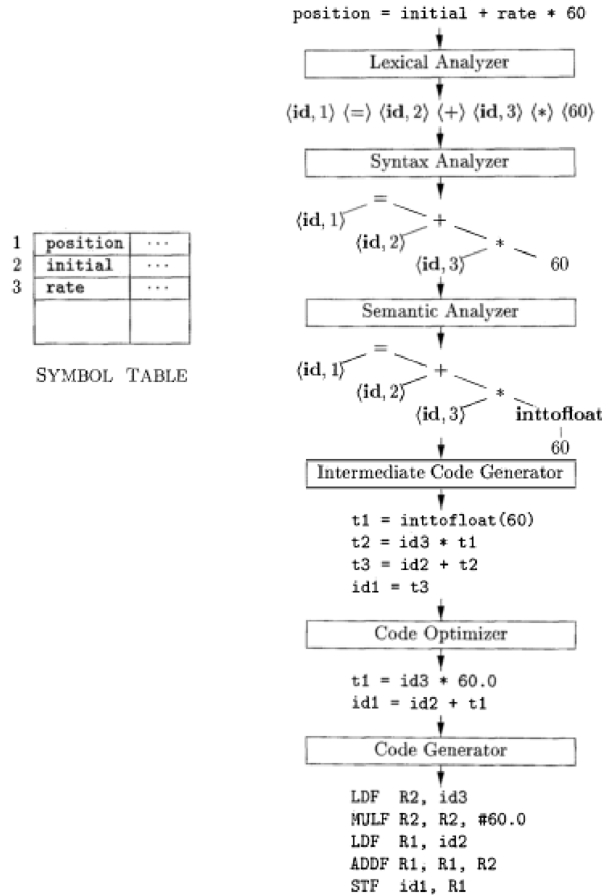

FIGURE 2.1. Translation of an assignment statement [1]

After the lexical analyzer, the compilation phase allows for the identification of the variable names, which can be defined in a symbol table. The lexical analyzer keeps track of the token names and attributes values [1]. The syntax analyzer gives structure to the lexical units produced by the lexical analyzer. Within the compiler process, related syntax bugs are visible within program grammar. When a program is at the semantic level, the compiler has a syntax tree that can be used to generate semantics about the program. A main focus of the semantic analyzer is to type check the supported language. The compiler verifies that the two value types are matched. Also, implicit type conversion events occur when needed. The convergence between numeral types is an example of these conversion events. Intermediate code generation takes in the semantic information and outputs and allows for the formation of machine-like code. Multiple analyzers can be used for this phase of compiler analysis.

Specifically, compilers such as Clang operate at the intermediate level and LLVM-IR code is a representation of the middle-ware, described by code language, that can generate the code. The intermediate code generated focuses on getting rid of redundant or inefficient code. Many programs at the semantic level of analysis are rooted in code optimization techniques, which are the next step in the compiler phase. This optimization occurs before the code generator phase. Code generation output data is produced for the machine to interpret and the results are binary and compile analysis. Binary code can be analyzed by Ida-pro and other tools; some compilers include parser generators. Most languages use a static scope and block structuring with precise control. Thus, programs can be broken down into modules, produces, and blocks. Compiler theories often subdivide blocks into different basic blocks and sub-blocks. With Clang, the blocks are further characterized as modules, functions, basic blocks, and instructions.

The compilation phases comprise important tools and representation. The parser generates the syntax analyzers from a grammar. Data-flow analysis has information on how values flow between components [15]. This is an integral part of code optimization. Many of these steps in a compiler occur in parallel. Most modern processors leverage instructions running in parallel. Many compilers have been influenced by RISC: a complex instruction set computer (CISC) [14]. For understanding compilation phases, it is also significant to address environments. Programs keep track of states of a program. These practices are so common in programming that many people do not realize that many languages build on these concepts.

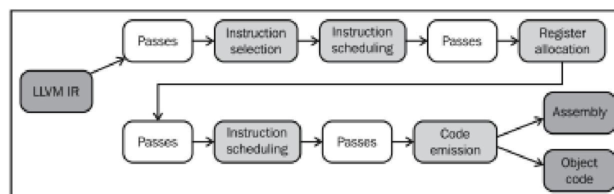

Figure 2.2. LLVM IR Pass Flow [31]

LLVM/Clang, also known as the LLVM project, is a collection of projects [31]. In this

document, the terms LLVM and CLANG are used interchangeably; however, Clang denotes the front-end compiler and LLVM is the back end. The LLVM project is the main compiler for many companies, including Apple. It is also is one of the more modern compilers that have a core set of libraries that translates source to a target that is, in theory, language independent. While CLANG primarily supports C and C++, developers and researchers can take a program in different languages and output a common, well-formed representation or machine code. That representation is known as LLVM IR. LLVM is heavy documented which is useful to those attempting to conduct compiler or any code analysis research. The Clang/LLVM project started off as a project at the University of Urbana-Champaign and is one of the top recognized compilers for C and C++. LLVM also has built in the LLBD library, which is a natively built de-bugger used to help debug memory issues. There are also many projects built using LLVM, such as LLVM Lint and Klee, used for analysis of programs for quality and coding issues. Though many people know LLVM for its support for C and C++, it also has support for many other languages, including Python, Haskell, Java, PHP and Pure for analysis using plugins. LLVM also supports standard libraries for C++ and is backward compatible with GCC. If a source can compile with GCC or G++, it compiles using LLVM. LLVM IR is used for research where programs are represented in the LLVM IR form before compilation. Figure 2.3 shows the benefits of using IR [38]. Clang is the front-end that generates the IR code. Another part of LLVM is the pass modules that plug into the application to perform different analysis. Passes can be used to inspect the code or to transform the code before outputting machine code. There can be many passes before machine code is generated.

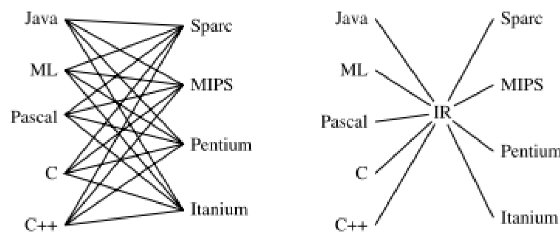

Figure 2.3. Source Code to Machine Code [38]

LLVM gives users the power to determine what functions, either call or callee, run in a program at any point during a pass. A function is a collection of basic blocks, which are similar to compilers' basic blocks; if a given basic block is called, all that code is executed. At the basic block level, LLVM also allows the user to see who the caller or callee is. The collection of basic blocks in the same module and function can be used to generate the control flow graph for a given function. Basic blocks contain instructions, which are actual single code operation similar to RISC code. The advantage of having RISC-like code is that operations have corresponding types and people can infer information on whether a particular code is a pointer related type or instruction. Another unique features of LLVM is that the LLVM IR form assumes an unlimited amount of registers. LLVM IR code is readable compared to machine code,meaning that programmers with little RISC or machine experience can read LLVM IR code. LLVM IR is expressive, typed, extendable, and well-formed. Like many languages, LLVM IR has opcodes such as add bitcast, return, and prime types such as void and i32. LLVM IR also has global and local identifiers. Global identifiers are functions and global variables begin with the @ symbol. Local identifications are register names and types that begin with %. Global variables and functions have different linkage types, which are important for users. Global values with a private linkage make it so the objects that can only be accessible by that module. Sample LLVM IR code is provided here [1].

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
%tmp = mul i32 %x, %y
%tmp2 = add i32 %tmp, %z
ret i32 %tmp2
}
```

[1]http://releases.llvm.org/2.6/docs/tutorial/JITTutorial1.html

2.7. Source Representation Analysis

There are some disadvantages to using LLVM-IR compared to other methods such as analyzing a program only at a lexical level. First, simple code in one language becomes very complex when represented in the RISC-like form. This is because things like complex expressions are extended throughout instructions. What might be achieved in one line of code might take several lines with control dependencies and several basic blocks to be expressed in LLVM IR. While LLVM provides the flexibility to compute data for different analysis, this analysis still requires computation time. LLVM uses algorithms to determine the best way to generate data for analysis; however, many techniques are still quite inefficient. Implementing advance analysis leads to more complex analysis on the source code, which can take longer depending on the complexity of both the code and algorithms used. There are two sides to this issue, which creates a trade-off environment. Either developers pick fast compilation and less accurate results, or slow compilation and more accurate results regarding checking for issues. Since computers are complex, getting exact, fast, and correct results is a challenging task. By default, LLVM only provides iterations over different groups: Modules, Functions, basic blocks, and instructions. Additionally, LLVM requires modules for analyzing code dependencies.

2.8. Limitations of Source Representation Analysis and Other Research

Many publications have focused on representing problems in different forms [12, 45, 50]. Dietz et al. found that compilers have different results when dealing with unsigned integer types. My study differs because the code properties can be used to learn from previous code patterns, including the behavior of the compiler when fixing these issues. Many SAST companies highlight one feature that is based on a specific representation. Yamaguchi et al. created what is known as a call graph [50].The call graph is very similar to the interprocedural graph, but contains more information. The authors put the information into the graph database and performed queries against datasets. They were able to create an automatic process for analysis, which is lacking in many other papers. Though the study's contributions to the field included search patterns and feature maps for source code, the

approach used by Yamaguchi et al. does not help when a programmer purposely adds a vulnerability into the application. The approach used in the my study differs because, in an instance in which a backdoor is patched, it is possible to learn from the code in terms of property graphs and to indicate other areas of the code that could be vulnerable. Sui et al. used LLVM to generate better pointer analysis [45]. They created an analysis tool that plugs into LLVM and took all the pointer locations and statements. They also provided a full suite of tools, which are available for analysis. The approach in my study differs because my focus is on finding issues, rather than just analyzing programs.

There are several drawbacks to the approaches used in the research mentioned previously. In both papers, the authors focused more on representation than on the severity and vulnerability of finding a bug in source code. It is important to show how different program representations can be used to find security bugs in applications. This is a critical part of any research on static analysis. Thus, my study highlights this oversight, while aiming to maintain effectiveness of finding issues in applications. Through using abstraction techniques, potentially vulnerable code locations can be found by extracting information from the system.

2.8.1. Program Representations

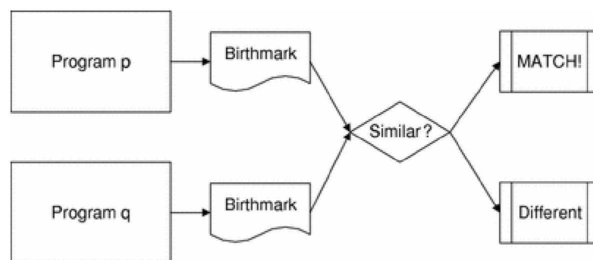

FIGURE 2.4. The Software Similarity Problem [5]

Clone code detection is a key aspect of this dissertation. Thus, it is important to review clone code concepts. The problem addressed in my study is a subset of clone code detection: the attempt to find program similarities between two code sections. My focus is on a smaller subsection of code. For example, if there are two programs, the goal is to

find a birthmark that not only indicates that these programs are similar, but also identifies that vulnerable codes share similar properties. Figure 2.4 shows the relationship between birthmarks and different programs [5]. The goal is to find distinct birthmarks in a project so that, if their properties are tested, it would be possible to determine whether the given birthmarks are matched or different. In the case of two different programs that have a similar birthmark, there is a match and those programs are the same. There are several definitions below that can explain these constraints [5]. These definitions can be used to formally express the problem involved with program matching.

Definition 2.8.1. Let r be a property for program p if for all possible executions r is true.

Definition 2.8.2. A program q is a copy of program p if it is exactly the same as p or it is the result of a semantic preserving transformation over p.

Definition 2.8.3. Program p and q are similar if they are derived from the same works.

Definition 2.8.4. Let p, q be programs. Let f be a method for extracting a set of characteristics from p . $f(p)$ is a birthmark of p , only if both of the following conditions hold.

- $f(p)$ is obtained only from p itself
- Program q is a copy of $p \rightarrow f(p) = f(q)$

Definition 2.8.5. Let p, q be programs or program component. Let $f(p) \rightarrow a$ and $f(q) \rightarrow b$ be the birthmarks extracted from p and q. Let $s(a, b) \rightarrow [0, 1]$ be a similarity function and a value $e < 1$. The birth marking system is resilient if p and q are similar and $1 - s(a, b) < e$

Definition 2.8.6. Let p and q be independently written programs. The software birth marking system is credible if the system can discriminate between the programs; that is $s(f(p), f(q)) < 1 - e$

DEFINITION 2.8.7. Given a set of programs and their classes $\{(p_1, c_1)...(p_n, c_n)\}$, the software classification function $c' = h(f(p))$ will yield a similar classification as close as possible to the true data set.

Software similarity can be divided into different categories: malware classification, software theft detection, plagiarism detection, and software clone detection. My focus is on a subset of software clone detection intended to find vulnerabilities in the source code. All programs have features which include syntactic and semantic features, such as raw code, abstract trees, variable pointers, and even instructions. Several definitions can be used to understand clone code detection or any area pertaining to coding birthmarks. If there is a program p with all the possible executions and q is a derivative of p, then there should be a formal equation that can transform the program from p to q such that $p' = f(p) = q$. The goal is to determine what $f()$ is in order to determine q. If p and q are programs, then p is used to infer a function such at $p \to f(p) = f(q)$. This means that, if the function can be determined, then there is an equation: $f(p) = f(q)$. If there is a function to compute the bounded similarity between p and q, then the values of that function would be between $[0, 1]$. If there is a variable e that is greater than $s(a, b)$, which is determined from the birthmarks, we can access the similarity of two birthmarks a and b. The difference between one function, say $s()$ so $s() < 1 - e$, and other functions can be determined. This means that $s()$ can also be evaluated. If there is the program p and the properties o, the similarity can be computed by a function $c' = h(f(p))$.

The following set of definitions address different properties or features in a program. It is best to start with the raw code because, at this level, the code is viewed as a string over the alphabet. This can be viewed as a syntactic feature. Similar to compilers removing comments for the syntactic feature, we remove comments in the code. The raw code would be nothing more than the symbols and variables allowed in the language. Thus, the raw code is all possible symbols allowed by the language in the alphabet. The abstract syntax tree can be defined as $r : P \to S$ so $p \to s, s, \sum$. Instructions are combinations of a given operand and any number of operands. Basic blocks contain a set of instructions (defined as

I) that can be used to execute operations. Each set of instructions makes up a basic block.

DEFINITION 2.8.8. Let $\sum$ be an alphabet of a symbol. The raw code of the program p is defined by the function r that evaluates to a string over the alphabet.

DEFINITION 2.8.9. Let I be set of all instructions such that $I = \{(opcode, opcode_1 .. operand_n)\}$

DEFINITION 2.8.10. Let InstSequence be a string of instructions such that $InstrSewuence \in \sum * \sum = I$

DEFINITION 2.8.11. Let InstSequence(b) be a string of instructions such that $InstrSequence \in \sum *, \sum = I$ for basic block b

DEFINITION 2.8.12. A program uses a set of procedures $F = procedures(p) = \{f_1, ..., f_n\}$

Basic blocks are used to make up procedures and functions that make up a program. Therefore, a procedure would be the set of all functions in the program. At this point, a simple program is defined as one that has no interconnections. While the functions and the makeup of functions are available, there is no way to connect the basic blocks or the functions together to make a program.

The next set of definitions relate to the control flow of a program. The control flow is a directed graph that shows the connections between basic blocks. Basic blocks show the flow of control between instruction sets and which basic blocks dominates over others.

DEFINITION 2.8.13. The control flow graph of procedure f is the directed graph $C = (B, E)$ such that B is the set of basic blocks and E is the set of edges between them.

DEFINITION 2.8.14. d dom n or node d dominates a node n if every path form the start node to n must go through d.

DEFINITION 2.8.15. A node d strictly dominates a node n if d dominates n and d does not equal n

DEFINITION 2.8.16. The immediate dominator or idiom of a node n is the node that strictly dominates n but does not strictly dominate any other node that strictly dominates n

DEFINITION 2.8.17. A dominator tree is a tree where each node's children are those nodes it immediately dominates

DEFINITION 2.8.18. The call graph of a program is the directed graph $CallGraph = (F, E)$ such that F is the set of procedures and E is the set of edges between them. The interprocedural control flow graph combines the control flow graph with the call graph. It is defined as $ICFG = (B', E)$:

- The set of control flow graphs
- Each control flow graph is given an additional exit node, which is a successor to the set of return nodes in the CFG
- For all basic blocks, a call instruction divides the block into two parts. the first part is connected to a call_return node, and that in turn is connected to the remaining basic block path.
- For each basic block that now ends with a call instruction, the block's successor is added the control flow graph of the call target. The successor of the exit node of the target control flow graph is additionally the call return node.

Control Flow Graphs are important in programs. All return statements in a CFG are terminating nodes. It is important to note that the path is a sequence of nodes in a CFG. The inter-procedural CFG is the CFG with call graphs. Control flow analysis also saves the execution order regarding program statements. All the possible execution paths create paths in the graph; multiple paths can create many programs. Recursion and other looping constructions can also mean that, for a given program, the amount of the time covered in a section of code can be large. CFGs show the flow between control given sets of instruction. Instruction sets occur when the control block is entered in the application. Basic blocks have one entry and one exit. For example, the program can exit a basic block only by executing

all the instructions in the basic block. The last node in a CFG is the exit node that will point to another node or could be the terminating node for the application, like the main function. The main function is a special function that exits and causes a termination. Edges show how the code will change the different flows, while nodes are the basic blocks. All paths in the program are exist in the program graph. With program dependence graphs, the goal is express dependencies between instructions or even blocks or code. In this case, instructions are considered. It is essential to show what instruction depends on other instructions for security purposes, but also in terms of program optimization and parallelization. A program dependence graph uses data and control dependencies. In terms of nodes on graphs, going between nodes that are dependent would show which nodes influence each other. Instruction data dependency shows how a value flows between locations in a program. The compiler generates correct code, detects illegal programs, and is involved with the management of the instructions. If there is an instruction or memory location that depends on another, that place has a data dependence on the other location. If there is a relocation, then that location no longer depends on the previous location. An advantage to using SSA forms in the source is that it is easier to discern dependence between values. If there is a chance of dependence between nodes, many techniques assume the existence of a relationship between data because register allocation is an NP-complete problem. It is also important to address the nuances of each program representation. The call graphs show functions and all the connections between nodes. They are popular in IDEs and can show recursion. Call graphs are different from CFGs because call graphs handle functions, whereas CFGs are isolated to functions. In the context of static analysis, all paths represent possible graphs. A control dependence shows if a location in the code will always occur before another location. All ways between that location have different relationships, such that items must come through. Strict dominance means that a location will occur before another and is not the same location. Further, strict dominance means that the two nodes are the same. Normal dominance implies a node's dominance over another and that the latter node can call itself. Immediate dominance means a basic block will occur immediately before another block. Post dominate is the inverse of

dominates; it implies that, if a program were to hit a given block, the program also has control dependencies: properties of graphs. Every path from Node 1 to the termination must go through Node 2. Control dependence determines that, if another value will execute, a node can also be control dependent on itself. The program dependence graph combines the control dependence graph and data dependency graph, which shows dependency and influence. Pointers are hard to handle in programs. Aliasing and ambiguity are bug issues involving pointers. LLVM has some basic functionality in terms of building pointer available information for a program.

Yamaguchi used a core graph to detect vulnerabilities using a pattern-based approach [50] and built a core graph representation. He combined the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Decesion Tree (DT), Program Dependency Graph (PDG), and Post Dominance Tree (PDT). They developed refinement parsing for their analysis and used the ANtrl4 for generating their custom parser. ANTLR is a parser generator. There are several drawbacks to their solution. Though it was very ambitious to build a novel grammar and intermediate parser, the trade-off was that their solution will only work with a limited amount of coding languages. Second, the SSA form provides more information not found in their approach. Using SSA is a key difference from my approach from Yamaguchi's approach.

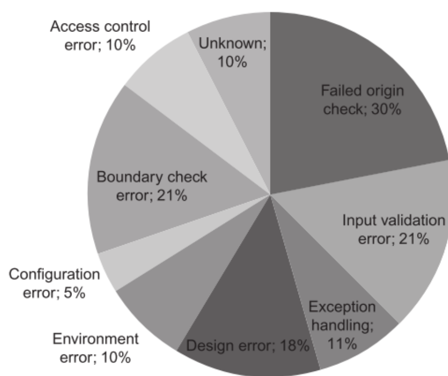

FIGURE 2.5. Vulnerabilities breakdown [18]

The following sections address different vulnerabilities. Guelzim et al. created a breakdown of different vulnerabilities [18]. This information is summarized in Figure 2.5,

which shows different vulnerabilities encountered in different applications. Failed origin checks and boundary checks occur in many applications. Configuration errors were the least prevalent, compared to the other analyzed vulnerabilities. He showed that applications have vulnerabilities and finding occurrences of those vulnerabilities is a difficult problem to address.

2.8.2. Null Pointer

Null pointers are a major issue for developers, and static analysis can detect some subsets of null pointer vulnerabilities [2]. For example, null pointer vulnerabilities can expose databases and cause numerous issues that can cause an application to crash [36]. My study is different because it contains analyses of different types of vulnerabilities, including null pointers. Null pointers have effects on even languages that are supposed to be pointer safe. Ayeawah et al. stated that "the Java coding standard recommends that if null is supplied for a parameter that is required to be non-null, a Null Pointer Exception should be thrown. Thus, if a de-reference of a potentially null parameter and an explicit throw of the parameter is null, both result in the same behavior (although an explicitly thrown exception might include a message that names the parameter that is null)" [2]. Many researchers confuse null pointer bugs with a potential for null pointer de-referencing. Having a null pointer is not a significant issue given some design flows. It is a widespread practice to have pointers that are initialized to null. Null de-referencing is the issue developers want to avoid. This is not a design choice, but rather an issue that leads to other issues in the system. In some cases, null de-references can only occur if a precondition is not set. In many cases, given all the properties of a given application, a precondition that yields a null pointer might not ever occur. However, common vulnerabilities in applications indicate that a case for null de-referencing is not reachable and that segment of code not causing dereferencing of a null pointer constitute two very different assumptions about the code. One concern with null pointers, as Ayewah et al. identified, is that many null pointer issues often persist through patch versions. Null pointers can persist in an application for a long period of time before they are patched. Hovemeyer et al. used static analysis tools to find null pointer bugs by

creating algorithms to see null pointer exceptions at runtime [23]. The authors were able to pinpoint 50-80% of defects involving null pointers. The present study differs because my framework includes null pointers and other vulnerability-causing software defects.

Basic analysis approaches involve looking for a null pointer detection without pointer analysis. Basic analysis approaches also involve an attempt to generate model values and look at control flows. Def use and aliasing approaches are used to extend the basic review to increase confidence concerning whether something is a pointer and thus should be dereferenced. Sui et al. created tools to analyze values and pointers [45]. They produced several different representations of the program so their analysis could generate inferences about the programs. They endeavored to show that programs that are heavily dependent on control and data dependencies had unsafe memory access. Hazardous memory access can cause bugs and unexpected behavior. Call graphs can be exploited when conducting interprocedural static value flow analysis, so they built techniques to review a program. SVF is a pointer analysis and value flow construction framework. The present study is different from the work of Sui et al. because code properties are used to capture data that can show the null pointer issue without having to analyze the def use cases.

2.8.3. Resource Drains

Resource draining in applications is another performance issue that can be exploited. This is a common way for DDOS systems. Once an attacker finds a resource that does not have controls in place to check values, he or she can exploit the system by constantly requesting for the resource in order to bring down the system. This will affect the availability of the system, which is a security concern. If an application is a developed while avoiding many runtimes and compile issues, the application can still have resource drain issues. Unbounded resources can lead to resource drain in applications. If a client can continue creating an object without boundary checks for that array of objects to an upper or lower limit, this will lead to a resource drainage issue. It is important to note that resource drain is very different from computational tasks that require significant resources to operate. Specific tasks can take a long time to generate results and require a lot of computational power. Resource

drain, compared to computational tasks, is a resource that, when exploited, can cause issues in a system.

2.8.4. Integer Overflow

	undefined behavior e.g. signed overflow, shift error, divide by zero	**defined behavior** e.g. unsigned wraparound signed wraparound with `-fwrapv`
intentional	*Type 1:* design error, may be a "time bomb" § IV-C3, IV-C9	*Type 2:* no error, but may not be portable § IV-C2, IV-C5, IV-C8
unintentional	*Type 3:* implementation error, may be a "time bomb" § IV-C4	*Type 4:* implementation error § IV-C1, IV-C6

```
1 int foo (int x) {
2   return (x+1) > x;
3 }
4
5 int main (void) {
6   printf ("%d\n", (INT_MAX+1) > INT_MAX);
7   printf ("%d\n", foo(INT_MAX));
8   return 0;
9 }
```

FIGURE 2.6. Number Handling [12]

Integer overflow bugs are hard to handle, hard to find in the source code, and can also cause fault errors in applications [12]. Figure 2.6 shows the different types of integer issues and an example of silent breakage [12]. Number handling issues are an assortment of problems involving overflows as well as underflows and truncation issues. Number handling, like many other issues, can lead to significant problems in an application. Many integer issues are well defined and are usually unique to a given language. Pre-defined integer type issues can be a considerable security risk. For example, the refactoring of Google's Native Client could cause 1 << 32 to be evaluated incorrectly with respect to security checks [12]. In the source code in Figure 2.6 , depending on the compiler, the conditionals would label INT MAX+1 as larger or smaller than INT MAX. This undefined behavior is bad for a system. Undefined behavior can create time bombs, in which the system randomly breaks on a given operation. Issues in the source code can involve integer overflow. There are both well-defined and undefined behaviors linked to intentional and inadvertent causes. Developers struggle to understand issues when applying algorithms involved with critical operations.

2.8.5. Number Handling

Number handling involves bugs such as integer overflow or integer overflow causing wraparound. Wraparound is not only assuming indexing into arrays, but also is when the

value should be more substantial than the initial or original value. This is an exploit or weakness involving taking advantage of calculations, which is a problem because an attacker can exploit it to gain control of the system. For example, if a check operation that requires integer boundary checking occurs before user input is used to run a given command, it can be exploited. These bugs can cause unexpected behavior that can have negative side effects. Many program controls use integer handling, such as loop controls. Further, defining memory allocation uses integer information to determine calculations. Different CWEs address several other issues involved with integer-related overflow and wraparound. For example, number handling can cause incorrect calculations and improper input validations. Improper input validation can create attack vectors for attackers. These consequences can lead to a high chance of infinite loops. Infinite loops have a high chance of also causing a resource drain. Resource drain is when the system keeps using a resource which affects the performance of the system.

2.8.6. Memory Corruption

Whenever a code performs operations in memory in an unintended way that leads to corruption, that issue is called a memory corruption. Some memory issues are pointer related. If a system has a buffer, that system can over-read or under-read data if the algorithms are poorly coded. While Android devices include Address Space Layout Randomization (ASLR) to reduce memory corruption attacks, the best way to prevent this is to find the locations in the code. Qin et al. expressed concerns about how memory management and user-related issues could be analyzed deterministically [43].This includes allocating memory in a location that can lead to corruption. They created a proxy system that took in system input to recover from a failure. In their Rx system, Qin et al. built a reporting system to handle reporting errors. They also developed methods for environment wrapping and rollback in case an error occurred. Although it is good to nullify the pointer after freeing memory, many developers code poorly, which can lead to memory corruption [52]. The present study differs from that of Qin et al. because there is no requirement for the end user to understand how memory corruption works, as long as they have some related historical patch data. In many

cases, isolating the root cause of the memory corruption is a difficult problem to address.

2.8.7. Concurrency Handling

Identifying concurrency issues requires two actions. When people consider concurrency issues, they generally only think of data races, conditions, and deadlocks [22]. Dealing with concurrency bugs entails several issues. First, concurrency bugs are hard to find. Any processing operation that can cause issues when running at the same time can cause a concurrency handling issue. These programs are also hard to maintain because of their non-deterministic interleaving usage in shared memory access [19]. There can be several threads using the same resource. Also, the advent of multicore processors has created an environment in which developers are encouraged to build applications that support concurrency. The second issue in terms of concurrency handling problems is that they are hard to reproduce. Also, researchers have focused on data races and deadlocks rather than other order violations [29]. Finally, concurrency issues are hard to fix as it is currently hard to find and reproduce bugs in large programs that are not found easily. When someone must deal with issues that require two or more operations to occur, it becomes hard to find in many applications. Many deadlock issues are never fixed. Lu et al. conducted a comprehensive study on concurrency bug characteristics [32]. Around 1/3rd of concurrent bugs use multiple variables to create a concurrency issue. More surprisingly, many of the bugs found were not fixed on the first patch.

2.8.8. Injection

Cross-side scripting and SQL injection, which are forms of injection, are critical issues with applications. These types of vulnerabilities are a problem, especially for web-based applications. The cause of this problem is that, often, developers do not think about developing an application securely. For example, when a team is developing a web application, many times, developers do not consider all the possible ways a program can be exploited. Query parameterization is a common way to stop attackers involved with user data. Without query parameterization, users can exploit server code and pass arguments to the database. Con-

versely, some programs have the ability to show information to a user who can then exploit the browser by using cross side scripting. Another popular form of attack is remote execution, in which the attacker's end goal is to run OS commands on the server to exploit the system. Some attacks include deserialization, which exploits dependent vulnerable libraries. Guelzim et al. stated that code injection and execution is when "an attacker alters the sequence of executed program instructions by injecting code at a specific memory location and altering indirectly a CPU instructor pointer to that malicious code region" [18].

2.8.9. Other Vulnerabilities

There are many other vulnerabilities that are not addressed in this dissertation. Further, many CWE reports exist concerning these different vulnerabilities. One common vulnerability is execution handling. Knowing when a program will fail or should fail an exception is an art. Exception handling is a subset of missing checks in the source code that leads to vulnerabilities in the application [51]. Often, manually auditing the application is necessary to detect a missing check in each application. Yamaguchi et al. built an application to check for missing checks in given applications. They were able to discover unknown application vulnerabilities in two applications using their methodology of looking at the source and detecting missing conditions. A common patching technique involves adding execution handling code to stop the exploitation of a vulnerability. However, this is often treating the side effect rather than the cause of the vulnerability. This can lead to the vulnerability never being fixed and attackers finding new attack vectors.

2.9. Literature Review

This section highlights differences between this work and extant research, and summarizes some of the key points addressed in previous sections. Several extant papers have focused on bug or defect predictors. One of the earliest published papers, by Menzies et al. [35], addressed how to learn defects. They found that the manner in which attributes are used to build the defect predictors is more important than the attributes being used. They were able to identify defect predictors with a 71% probability of success. This was

better than the IEEE evaluation of manual software review, which suggested that people could find, on average, 60% of the bugs. McCabe focused on using a graph-theoretic complexity measurement to show how to understand program complexity. He also looked at different approaches for building feature spaces used to predict defects. By using different combinations of filters and learner algorithms, he was able to achieve a detection of around 71% with a probability of false alarm at 25%. McCabe's study found that defect predictors should be built using all attributes rather than focusing on a subset of attributes for a given domain. The present study differs from earlier research because different features are tested for during feature extraction as expansion of the number of types of models are tested for analysis. This allows for the exploration of both properties and learning algorithm configurations. Neuhaus et al. [39] addressed bug prediction for Mozilla using machine learning; uniquely, they built classification labels and determined whether a file was vulnerable. They built their feature space on the import/include statements found in each file and evaluated this using an SVM. Out of all the vulnerable functions used to evaluate the model, their system detected and labeled 45% of the files as vulnerable. For all the components flagged or labelled as vulnerable, 70% were vulnerable. This means that the system misclassified 30% of components. Neuhaus et al. focused on the module, which is a limitation because it will not find where the vulnerabilities are found in the function. Thus, it can only be generalized to the class file. Further, their work does not address when a file's vulnerability is patched and they did not cover how to address whether the function was patched and the effects it would have on the model. Mens et al. conducted one of the first studies to introduce the use of historical data to predict bugs in an application [34]. Their work noted that some modules are more valuable than others and focused on understanding the impact of problem domain, code complexity, historical data, and several other areas affecting a defect predictor. They suggested that complexity, problem domain, evolution, and process are key factors that make programs defective. They found that complexity metrics correlate with defects, but there is no well-defined universal metric. The present study is different because different models and approaches are combined to gain understanding of predictors used.

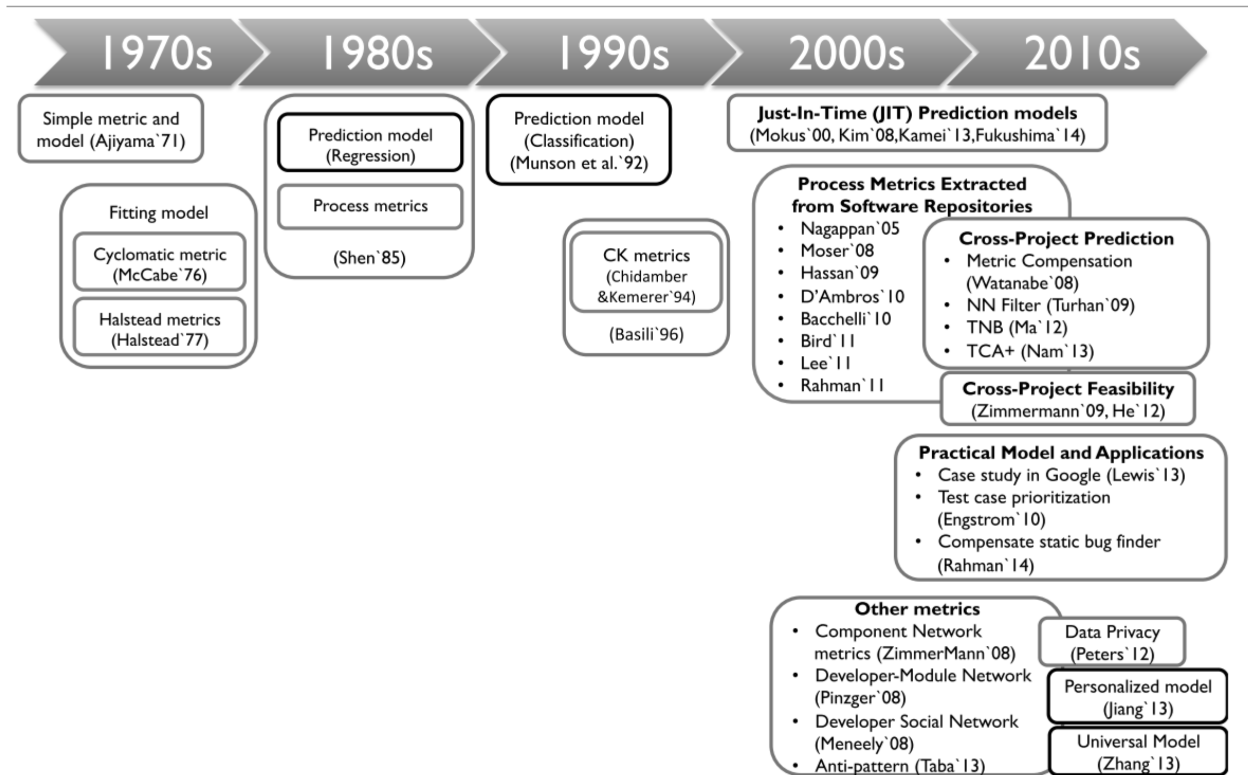

Figure 2.7. History of software defect prediction studies [37]

Figure 2.7, in Nam's study, chronological review includes more recent publications [37]. Nam surveyed current approaches since 2014 and presented the issues involved with future research [37]. Nam reviewed the defect prediction process and techniques used, including metrics, models, and algorithms, across several papers. Figure 2.7 shows all the different approaches Nam identified, including the categories used for software defect prediction. Nam found that the algorithms covered in other papers included classification, regression, active learning, and semi-supervised learning. Nam identified the most common measurements used to evaluate different prediction models are F-measure, recall, and area under the curve (AUC). An overwhelming number of researchers used size as a default predictor. CK (Chidamber & Kemerer) size, object-oriented features, and McCabe features were the second most popular defect prediction metrics using frequency. Classification and regression techniques were used in most papers with logistic regression; Naïve Bayes, and decision trees as the most frequently used models. My study not only considers these models but also includes

cross prediction using different programs available in the testing datasets and expands on the analysis process.

D'Ambros et al. created a meta-model for handling software artifacts and implemented their framework and analysis techniques [9, 10]. They noted that bug prediction can be based on changes in metrics, earlier defects, code metrics, entropy, and churning of the code. They found that using features based on CK size and object-oriented metrics was the best approach, in terms of data, when using Spearman's correlation coefficient. They found that bug prediction based on a single metric did not work well and that the best weighting was achieved by using past information from the application using linear models. The present study differs because my focus is at the function level, rather than looking at all the files. Focusing on the functions enables a greater granularity for analyzing applications. Though object-oriented program properties are important, the solutions developed using these properties will not work for non-object-oriented program paradigms.

Nguyen et al. used historical data in addition to dependency graphs to predict where the vulnerability is in a program [40]. These dependency graphs were based on the relations between components, functions, class, and variables. They tested their dependency graph predictors using the JavaScript Engine used by Firefox. They tested several classifiers including Bayesian Network, Random Forest, Support Vector Machines, and Naïve Bayes. They used information generated from Doxygen, a documentation tool, to find vulnerabilities in components. The present study differs because it fits into the build process of an application. Another key difference is that code property and information generated by the compiler were used to improve the model performance.

One of the first papers to introduce text mining for bug prediction and the use of android applications was published by Scandariato et al. [44, 47]. The work of Scandariato et al. is different from other papers which focused on desktop and web-based applications because the authors found that they could create prediction models for Android applications and that prediction technique can forecast reliable performance in vulnerable files for later versions of Android applications. It was also the first paper to show that some models

in applications can also work in a different applications. In the present study, different applications were found and performance was compared to other applications. Comparisons between several models were made to evaluate the performance when testing against different applications.

Wang et al. introduced the idea of using semantic features to learn defects in a given application [48]. They used vector tokens based on the Abstract Syntax Tree and belief networks as the learning model. Their study showed that using semantic features can work for bug prediction. Their approach differs from the one used in my study, which is based on the IR form. This not only makes my approach application independent, but also language independent. Wang et al. proved that semantic features can help bug prediction. The improvements in their study were based on semantic features and not on a classification algorithm. Pianco et al. sought to understand what could be learned from patch history [42]. Though their research did not analyze the program using learning techniques, they were able to differentiate between vulnerable and non-vulnerable code using information based on change history. Unlike Pianco et al., in the present study, the cause of the vulnerability is determined using several learning algorithms to evaluate performance. Pianco et al. did not adequately analyze the performance of using learning models and instead used a high-level approach. For testing against Eclipse, the support vector machine tested had 67% precision and random given defect packages of 37%. They also found that the more code changes that occur in a given part, the more likely that part is to have issues. However, they did not test the results against other applications and using different techniques. In my study, we consider multiple applications and techniques..

2.10. Performance Metrics

Many metrics are required in order to analyze the performance of different models and classifiers. Common metrics used include precision, F1 measure, recall, and accuracy. These metrics give a single value that represents model performance with respect to a contingency table. A more general name for a 2 by 2 contingency table is a confusion matrix, which shows the total population. Each cell shows a predicted and true condition. Many metrics

are calculated using the contingency table. The confusion matrices show the outcome with respect to the table. A true positive means that the model classifies an item properly. This mean that the item is classified in the correct group. A false negative means that the model classified something as false when it is true. This is bad for any classification of vulnerabilities as a presence of vulnerability is classified as non-vulnerable condition. A false positive is when a model classifies something as positive when it is not positive and was instead supposed to be classified as false. Many times, this is referred to as a false alarm, in which someone is alerted when nothing is wrong. A true negative is when something is classified as negative when it is actually negative.

Metrics such as precision, recall, and F1 score can be built against a model using the contingency matrix. The precision is how many items are selected, while recall is the number of relevant items that are selected. Precision is calculated by the number of relevant retrieved items over the number of retrieved items. The recall is the number of the relevant retrieved items over the number of relevant items. Formally, accuracy is the proximity measurement of the results based on the actual values. Accuracy shows the systematic error and is the number of correctly calculated items over the sample size. This means that the accuracy is calculated by the number true positive and true negatives over the number of true positives, true negatives, false positives, and false negatives. The F1 score is the final metric used. It is the harmonic means between precision and recall ($F_1 = \frac{(2 \times precision \times recall)}{precision + recall}$). F1 score is a balance between precision and recall.

CHAPTER 3

DESIGN AND PROCEDURE

The objective of my study was to find vulnerabilities involved with historical source code information. The area of this study is categorized within areas such as bug prediction and as a subset of clone code detection. This section includes information on the datasets involved in the evaluation of the approaches. The next section includes a discussion on the core graph based on LLVM IR code. The core graph is a vital part of this research. The next section also includes the novel approaches created using rules and other historical data to decide diverse types of vulnerability classes. This section also covers different approaches for finding patterns to detect vulnerabilities in a given application. The contributions of this study are:

- Construction of a core graph using LLVM CLANG IR
- Framework for evaluating code based on code properties
- Several novel historical approaches for detecting vulnerabilities in applications based on historical data augmented with learning techniques

3.1. Hybrid TF-IDF for Vulnerability Detection

A hybrid TF-IDF approach was created, which is addressed throughout the methodology and results sections. This hybrid approach was used because of the need to summarize vulnerable functions in a few core properties. While these properties are features, all the information used is not as important as other properties. Alternatively, an approach that used only the frequency based on the number of instruction type pairs and edge relationship could have been developed. However, specific instruction types and relationships occur frequently in a given module, which can cause issues with the model's performance. Thus, I chose a hybrid approach, which takes the Cartesian product of the instruction and relationship pair and multiplies it by the unique pairs in the module. This gives the vulnerable function a unique marker, rather than looking at the most frequent operations.

TF and IDF can also be viewed as two different scoring metrics that use the dot product. The tf score shows how important the instruction type or instruction relationship is to a function, but does not show how its proportionality to the frequency of other instructions and relationships shown in the dataset. Consequently, the inverse document frequency is calculated because it because it adds a collection of ranking and weighting to the current frequency. This allows rare items to become more relevant based on the high weights for the combinations of ranked frequency.

3.2. NIST SAMATE

The NIST SAMATE is a collection of projects used to evaluate the efficiency of static code analysis tools. While, in this document, I use NIST SAMATE to refer to many aspects of that collection of programs, it took several teams to build all the tooling chains. Over the years, third-parties have also contributed to the datasets. The NIST SAMATE group hosts the data repository involved with Stonesoup T&E. Stonesoup stands for Securely Taking on New Executable Software of Uncertain Provenance (STONESOUP), and was a project funded by several groups with the goal of neutralizing different weakness types in programs. Some of these groups focused on looking for coding flaws and protecting applications automatically. The Test and Evaluation eXecution and Analysis System (TEXAS) is the command line component that interacts with the datasets.

The NIST Samate tooling environment offers many benefits because it provides a means to evaluate an approach while maintaining the integrity of the application under test through application build up and build features. The NIST test environment can plug into this other security tools by changing the build process and environment variables. The NIST Samate tooling environment is one of the most complete testing environments for static analysis solutions. NIST Samate includes a collection of vulnerabilities that developers and application vendors can evaluate their approaches against. The NIST Samate project's goal was to reduce the number of vulnerabilities by improving assurance, which was accomplished through the development of techniques that allow for software evaluation and measure the effectiveness of current tooling. Another goal was to identify gaps in applications, and

Samate's scope ranges from source code security to SCADA systems. The project has gained users from developers and large code analysis vendors. The key goals of the project were to enable research on large test sets, improve tools, and improve tool adoption [1]. The project was proposed by the Department of Defense, and began in late 2004. NIST maintain the software assurance reference dataset (SARD), which is a community-driven repository of vulnerabilities artifacts. As of late 2016, they had more than 170,000 test cases in different languages with different flaws.

The main goal of the Samate research was to neutralize vulnerabilities in software and address common implementation weaknesses in applications. They developed a three-phase process by which to neutralize the percentage of vulnerabilities found in different SDLC programs. Another key difference in Samate's tooling compared to other tooling is that their focus was on analyzing programs rather than the result of the problem. This involved looking at what caused the flaws rather than the vulnerabilities in the given applications. The goal was to apply protection and patching without requiring developers' intervention. One of the future goals for the tooling was to be able to track user input at runtime, which is now available in some RASP solutions. Figure 3.1 shows information corresponding to the different vulnerabilities tested using my approach:

Identifier	Base Program	Category	Version	Repository	LOC
FFMP	FFMpeg	Console	1.2.2	https://www.ffmpeg.org/	566,908
GIMP	Gimp	GUI	2.8.8	http://www.gimp.org/	711,339
OSSL	OpenSSL	Console	1.0.1e	https://www.openssl.org/	274,204
PSQL	PostgreSQL	Service	9.2.4	http://www.postgresql.org/	731,469
SUBV	Subversion	Console/Service	1.8.3	https://subversion.apache.org/	798,636
WIRE	Wireshark	GUI	1.10.2	https://www.wireshark.org/	2,523,396

FIGURE 3.1. Samate Programs [11]

Figure 3.1, shows some of the programs included in the Samate dataset. These are not all the programs, but rather some of the critical applications included in the dataset. The programs ranged from services to GUI-based applications. Most GUI-based applications have

[1] https://samate.nist.gov/index.php/Introduction_to_SAMATE.html

similar functionality in the back end when comparing applications. Versions of applications provide a baseline, which allows static analysis tools to run against the same application. The applications' size ranged from a few lines to millions of lines of code.

Many of the applications are real-world applications that are available online now. For example, FFmpeg is a tool used to play and convert mp3 files. It used several other programs to extend those applications' support. GIMP is an open source application used for photo editing. Many artists use GIMP as an alternative to Adobe Photo Shop and other image processing tools. OpenSSL, which is a key application of interest for this work, is a cryptographic toolkit. It includes many security tools involved with TLS and SSL. It is commercial grade in design and functionality, compared to many other tooling. It is also a general-purpose cryptography library. There is no UI for OpenSSL without using a third-party tool or building a wrap around. Subversion is a version control tool. Finally, Wireshark is a packet monitoring tool used by security specialists to watch network traffic going between a device and the application.

Complexity Feature	C	Java
Control Flow	13	17
Data Flow	17	6
Data Type	7	3
Total Phase 3 Complexity Feature Permutations	13 x 17 x 7 = 1547	17 x 6 x 3 = 306

FIGURE 3.2. Samate Number of Complexity [11]

Figure 3.2 shows the different program properties. The complexity features include control, data flows, and different data types. Even though it might seem easy to use only control and data flow, these are common properties that make up a program. Any application that has any meaning will have some operations involving control and data flow. The combination of distinctive features creates an enormous range of different test applications that many static analysis tools can be evaluated against. Over 1,700 different vulnerability classes are generated for the application. In general, the applications focus only on one vulnerability; there can be any number of these types of vulnerabilities in a given application. One real application code often has several vulnerability classes in various locations and

modules. This implies that tools or algorithms used on this dataset can easily extend their applicability to real-world applications.

Weakness Class	# CWEs	
	C/Binary	Java
Number Handling	9	8
Tainted Data	N/A	6
Error Handling	N/A	8
Resource Drain	11	9
Injection	3	4
Concurrency Handling	15	15
Memory Corruption	17	N/A
Null Pointer Error	1	N/A

FIGURE 3.3. Weakness Classes [11]

Figure 3.3 shows the makeup of different vulnerabilities. In the Samate datasets, these are referred to as weakness classes. This section includes a review of the different classes in the Samate dataset. One fundamental difference is that not all classes capture concepts in different languages. There were no vulnerabilities recorded as using tainted data and error handling involved with C and binary applications. Applications other than the one illustrated in Figure 3.3 have vulnerabilities as well. Applications in the core group of applications were not found to have any issues with memory corruption or null pointer errors. The weakness class corresponds to the CWEs related to the application; however, this does not necessarily mean they make up the total number of applications in a given test suite.

- Injection
  - CWE-78,CWE-88,CWE-89
- Concurrency Handling
  - CWE-363,CWE-367,CWE-412,CWE-414,CWE-479,CWE-543,CWE-609,CWE-663CWE-764,CWE-765,CWE-820,CWE-821,CWE-828,CWE-831,CWE-833,
- Number Handling
  - CWE-190,CWE-191,CWE-194,CWE-195,CWE-196,CWE-197,CWE-369,CWE-682,CWE-839,

- Resource Drains
  - CWE-400,CWE-401,CWE-459,CWE-674,CWE-771,CWE-773,CWE-774,CWE-775,CWE-789,CWE-834,
- Null Pointer
  - CWE-476,
- Memory Corruption
  - CWE-120,CWE-124,CWE-126,CWE-127,CWE-129,CWE-134,CWE-170,CWE-415,CWE-416,CWE-590,CWE-761,CWE-785,CWE-805,CWE-806,CWE-822,CWE-824,CWE-843 Access of Resource Using Incompatible Type ('Type Confusion')

3.3. Core Graph Representation

A large part of this research focuses on program representation. I based the development of the core graph representation of each program in this study on SSA in LLVM IR. The core graph is the graphical representation of a program, which includes the program dependency graphs and the ICFG graphs involving an application. A similar study was conducted by Yamaguchi et al. [50]; however, it was unlike this study because my implementation is based on SSA from LLVM IR, which makes this approach closer to language independence. Additionally, I expanded this study to take advantage of functionality added to LLVM through its release cycles. Another advantage of this study is that my approach can easily plug into the build process, whereas the process developed by Yamaguchi et al. requires an independent compilation process for the generation of results.

Given that this study relies on LLVM to build the initial SSA form, its compilation cost is higher than other approaches that rely only on the raw source code. Any non-linear addition to the graph or analysis could have significant computational time compared to other approaches. This is a concern, but it does not take away from the merit of this work. Many static analysis techniques take a long time to analyze source code; all these techniques, to best of my knowledge, suffer from similar issues. Secondly, the graphs can be exported nightly or over the weekend, which aids in running analysis without slowing down the development. This provides a flexibility that earlier research methodologies did not have.

I used several libraries to analyze the graphs. Given the dot language, even the graphs themselves can be used by different programs or tools that can parse the dot language. One of this study's goals was to strive for language and analysis independence for later research that would have a cost in computation or analysis time. Thus, I developed an archiving system for handling the vast amount of different program files involved with the build process. This work relied on the file structure with the commit hash to decide what changed between the files. Additionally, I stored all source code as bitcode and dot files in the corresponding files for improved search speed, while trading off on build time and storage. By permutating the file extensions, it was possible to easily decide whether a c file was built, and what the other output files were. If a .ll file was not found, this meant that there was also no dot file in the same folder. This process, though expensive on the file system, made it possible to run analyses more efficiently, compared to building the program each time.

3.3.1. Program Structure and Composition

I investigated several applications using the sample dataset. Table 3.1 shows the application extracted for the purposes of my study. This study did not address, in depth, why some programs did not compile, because there could be several environmental issues with the system, which is beyond the scope of this dissertation. Below in the table shows the makeup of the different applications. There two parts to this research: an analysis on using machine learning concepts against the SARD dataset and analysis on machine learning concepts against real-world OpenSSL application. For the purposes of the NIST part of this study, my focus was on PSQL, OSSL, GREP, CTRE, and FFMP.

In Table 3.1, different numbers of application versions were able to compile, which have different vulnerability classes. OpenSSL has several types of vulnerabilities. Altogether, I pulled more than 4,000 test cases from Samate across different applications. I then reduced the number of test cases to more than 2,000. Each test case corresponding to the CWEs had at least 15 related test cases. Some CWEs, such as CWE-476 and CWE-089, included more than 100 cases. For the analysis, I used all the SAMATE applications shown in Table 3.1. Each test case had different variants. The variants were based on the CWE, program,

Application	Used in Testing	Application	Ignored
PSQL	637	SUBV	638
OSSL	636	WIRE	637
GREP	380	GIMP	637
CTREE	380		
FFMP	637		

Table 3.1. SAMATE Applications

and various program properties. This included taint source, data type complexity, data flow complexity, and control flow complexity. There were also different increments associated with each test case. These increments were used to stop collisions. There were 10 types of cases that used the unique increment. The incremented counter shows the developer generated vulnerabilities for test cases. In Table 3.1, most of the types were about 100, given all the test cases analyzed. There were several injection points used throughout the programs. The injection points are specific to the program. The ID used can be same; however, given the code properties, they can refer to various locations in code. For example, for the OpenSSL test applications, there were around 160 different injection locations throughout all the test cases. The taint source (TS) showed the area of the general cause and location. This is the location affected by the vulnerabilities. Different vulnerabilities can affect various systems in diverse ways. There were four types of test cases: environment variables, file contents, sockets, and shared memory. Java, C, and binary could have any of these areas that reflect a given vulnerability. Most of the weaknesses in the OpenSSL test cases were C088A, C476G, C476D, and C476A. Figure 3.4 shows the distributions of different vulnerabilities generated at different levels in the code: injection point(IP), data type(DT), data flow(DF), and control flow(CF). It shows the variability between the total number of classes of properties. For the case of OpenSSL, there was an elevated level of variability dealing with the injection points.

To summarize the OpenSSL data, I generated different figures related to the properties with the CWE occurrence information. Thus, I made a pair point graph using the CWE,

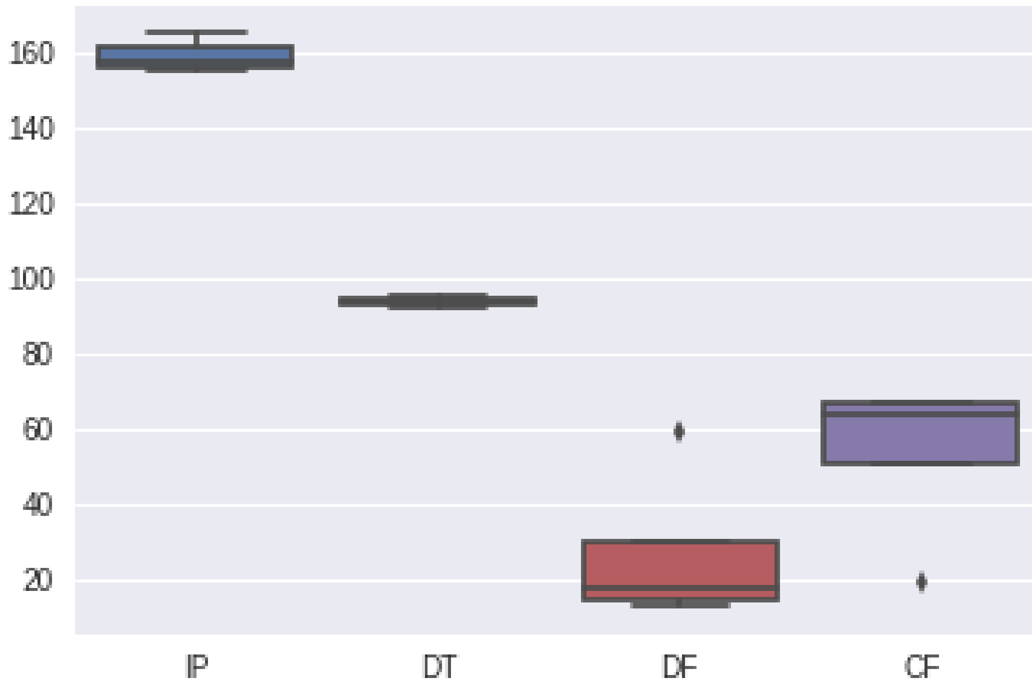

FIGURE 3.4. BP SM OSSL TS Count

IP, TS, DT, DF, and CF. The pairwise graph shows the relationship between each program property given the CWE; the diagonal information displays the one property given the different CWEs. Though their CWEs are not in any given order, the HUE changes still show the changes between CWEs. The graph shows how the types of CWEs encountered in each test case change and the related properties change as well. This means that vulnerabilities for the same dataset indicate a relationship between the properties and CWEs.

Given the complexity of the dataset, I also analyzed the clustering of the data. I generated the Paired Density and Scatterplot Matrix, which shows the density relationship between pairs of program properties. Figure 3.5 illustrates separation between the CF and DF properties. With the distinctive features, Figures 3.5 and 3.6 shows that the properties can cluster together. I identified a large number of clustering between properties. This means that several properties do have some relationship to each other.

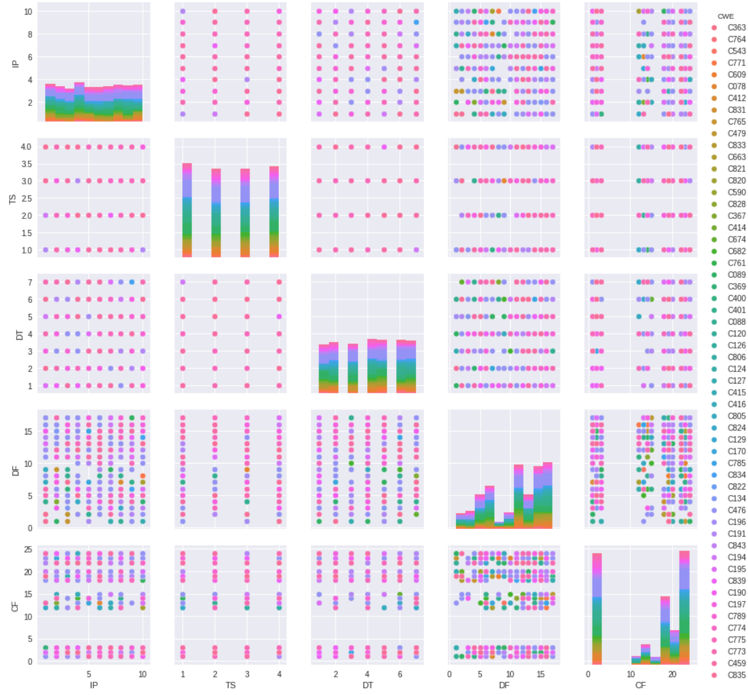

FIGURE 3.5. PP SM OSSL TS Features

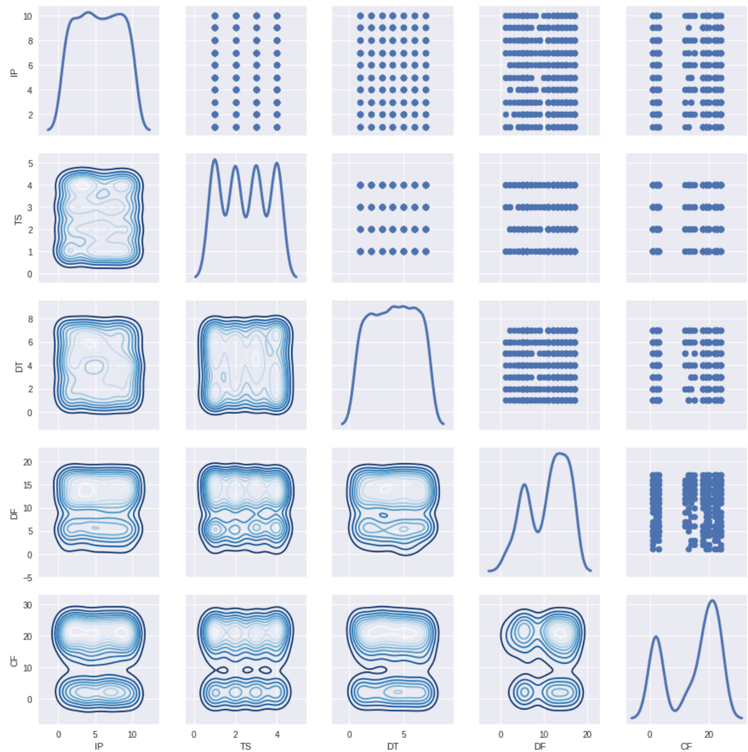

FIGURE 3.6. PG SM OSSL TS Features

3.4. System Configuration

For the environment, I set up a workstation machine running Ubuntu 16.04 LTS. The reason I chose to use a workstation over using a standard device was that the construction of graphs and computation involved with calculations for code analysis was computationally intensive. The configuration of the machine consisted of the following features:

- Machine - Ubuntu 16.04 LTS
- Memory - 62.8 GiB
- Processor - Intel Xeon(R) CPU E5-2670 at 2.60GHz x 32
- OS Type - 64-bit

3.5. Current Version of OpenSSL and Commit History

OpenSSL, one of the most popular cryptography toolkits used by developers for TLS, has 131 known vulnerabilities listed in the National Vulnerability Database (NVD). Of all the known vulnerabilities of OpenSSL, 72% were found after 2009. One of these vulnerabilities was the heart-bleed attack, which allowed hackers to access sensitive data from 24-55% of popular websites [13].

To help mediate the risk and prevent future vulnerabilities, a great deal of work is being undertaken on different techniques, such as using control flow graphs to detect vulnerabilities or generating malformed certs to detect when a TLS implementation will accept invalid certificates [4, 21]. While both papers [4, 21] focused on finding new vulnerabilities in different implementations of TLS, neither tried to find the root causes and similarities between groups of vulnerabilities. Given that many of the implementations of TLS are recompiled versions and analyses of the past actual vulnerabilities, this shows, at a minimum, a signature of different exploits and, in the best case, a repeating pattern. Additionally, many vulnerabilities are officially recognized by several engineering groups such as the MITRE Corporations Common Vulnerabilities and Exposures (CVE) and NIST's National Vulnerability Database (NVD). By backtracking through different resources and implementations, I found and analyzed historical versions of applications. My findings indicate that, out of 90

Last Commit Hash	3503549ee8bd59d23d00b9dbbc2444e91fc44746
Commit Date	July 8 2016
Version	1.0.2l
Home Page	https://www.openssl.org/
Download Link	https://www.openssl.org/source/

Table 3.2. OpenSSL Version Information

commits to fix exploits in OpenSSL, 83% occurred in a single file or function and more than half of the vulnerabilities were direct error handling exception bugs.

Since many of the packages are re-compilations of OpenSSL, it is essential for internal developers and code reviewers to be aware of these occurrences. This is relevant because the results found on error handling exception bugs are attack unspecific. I extracted data from the NVD database[2], OpenSSL source code[3], and from OpenSSL's homepage about CVE vulnerabilities[4]. Since the NVD database builds on the CVE with information such as the products affected, I used NVD. Using scripts, I conducted a search for the listed vulnerabilities on OpenSSL referring to CVE vulnerabilities, and found a total of 131 entries. I then searched the NVD for the related commit files for each vulnerability. Since a part of this study involves analyzing the source files, I extracted all the products affected. Initially, I found 56 reports. After further analysis, I obtained 34 additional commits by reviewing the descriptions and meta-data in the NVD database, OpenSSL site descriptions, and advisory reports.

I used the descriptions on the vulnerabilities from the NVD database, information from OpenSSL advisory reports, and descriptions in the commit files for analysis. I collected 17,231 commits from OpenSSL source code [5]. Table 3.2 provides summary information on

[2]https://nvd.nist.gov/

[3]https://www.OpenSSL.org

[4]https://www.openssl.org/news/vulnerabilities.html

[5]Links last checked: 10/11/2018

the OpenSSL version used for this study. With the commit hash, the repository can be reversed back to a different version of OpenSSL. Until 1999, no commits to the source codes were found mapping to CVE reports. This may be due to the poor reporting practices and a lack of coordination with large vulnerability databases. All five of the first vulnerabilities were buffer overflow related. These range from flaws in ASN1 to issues with Kerberos implementations. Prior to 2008, the ratio between file changes and the commits for each vulnerability was high. This meant that the number of files changed in each commit was high when repairing known issues. I found reports after 2008 that contain most of the vulnerabilities.

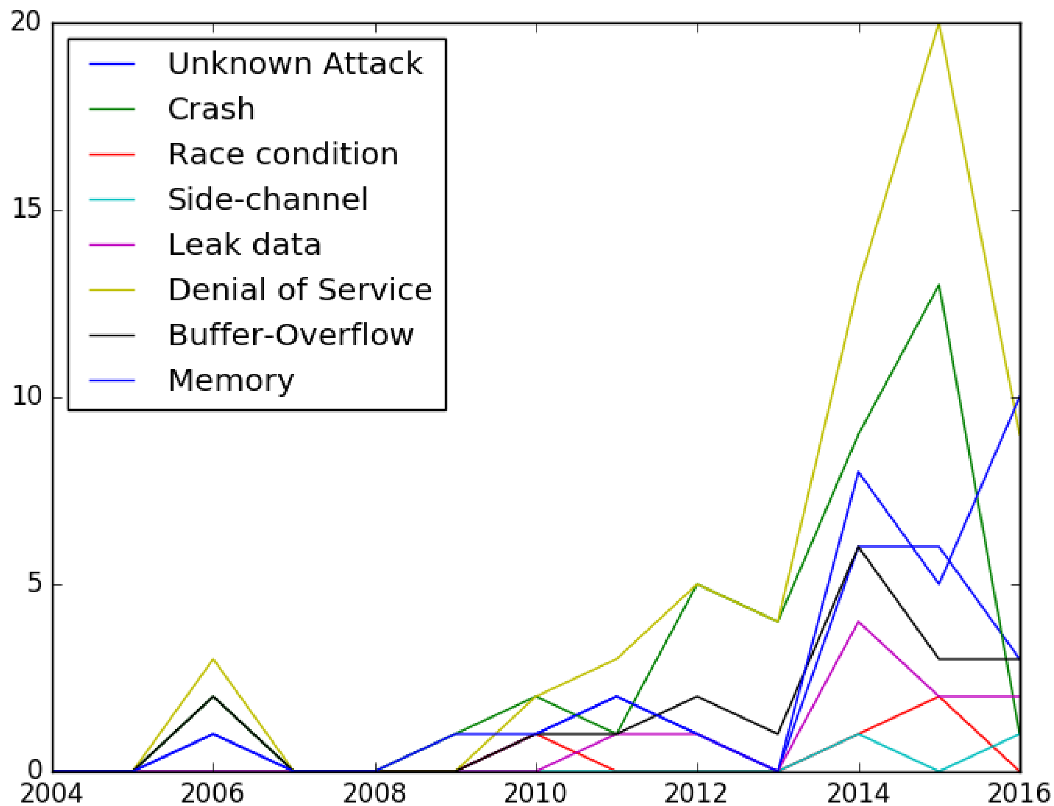

FIGURE 3.7. Attacks Through Time

Figure 3.14 shows the trend of CVE reports over the years in OpenSSL. After 2012, there is a positive trend with fewer files changed in the commits to fix major vulnerabilities.

For the purposes of this study, I divided the vulnerabilities into overlapping groups based on the analysis of the meta data and source code. Figure 3.7 shows several types of attacks over the years. Unknown attacks refer to any attack that could not be classified properly into any of the other groups. This means that either the CVE for the attack is not specific, or it is unclear how to determine the cause of the attack. A crash vulnerability, as the name implies, causes the program to crash. For example, CVE-2006-3738 is a buffer-over-flow issue in the SSL in which shared ciphers caused the program to have an overflow. When a client connects to the malicious SSL server, it could crash the client. A race condition refers to any condition in which two operations should not have occurred at the same time. Race conditions occur when dealing with multi-threaded operations. Side-channel attacks are any processes that gave information on implementation. Usually, these vulnerabilities occur when people try to improve performance through code optimization. Leak data, through any attack, is any information able to be captured. These could be buffer overflow or buffer underflow attacks. Buffer-overflow is any program that overwrites the boundary of a given buffer. A denial-of-service (DOS) attack can create a buffer-overflow. A DOS attack is any attack that causes repeated connection to a system. Memory-based attacks are any memory related attacks that rely on pointer issues that affected the memory. If several attacks were related in the code and description, they are counted as different vulnerabilities.

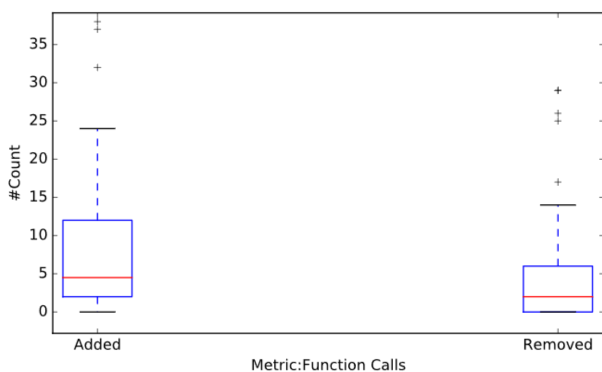

FIGURE 3.8. OpenSSL Function Calls in Diff Changes

I analyzed each vulnerability diff file to assess how the files were repaired for any vulnerability in order to show if more complexity was being added to the codebase. Figures3.8

and 3.9 show the changes in the files and how many new lines were added or original lines were removed. The number of lines of code increases when functions are modified. This finding was independent of the type of vulnerability.

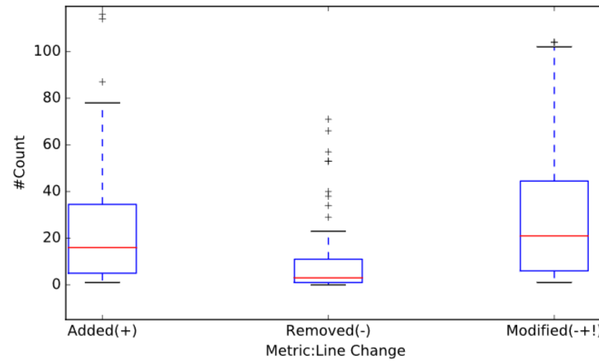

FIGURE 3.9. Line Modifications in Diff Changes

Figure 3.10 shows the difference between authors, reviewers, and committers when patching issues in an application. To keep the users' personal information private, I converted the actual names into IDs for the corresponding user. These figures show the meta complexity of a given application. Most of the code changes are performed by a few people. For each graph, two people account for more of than 50% of each pie chart.

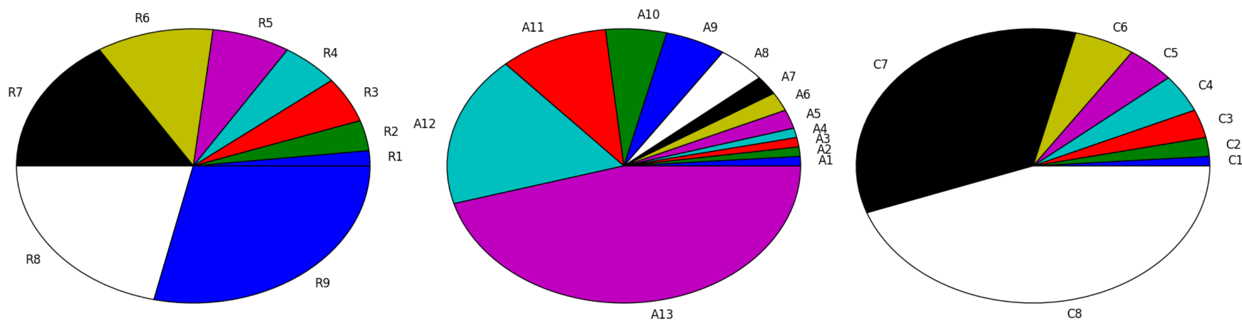

FIGURE 3.10. OpenSSL CVE: Authors vs. Reviewers vs. Committers

Using control flow graphs is a very popular way to detect vulnerabilities in source code. I used the graphs to perform static analyses on code to tell key differences. The graphs show all the paths that can be traversed through a program during run-time as well as the relationship between reachable calls. I extracted all the c files, before and post commit, to

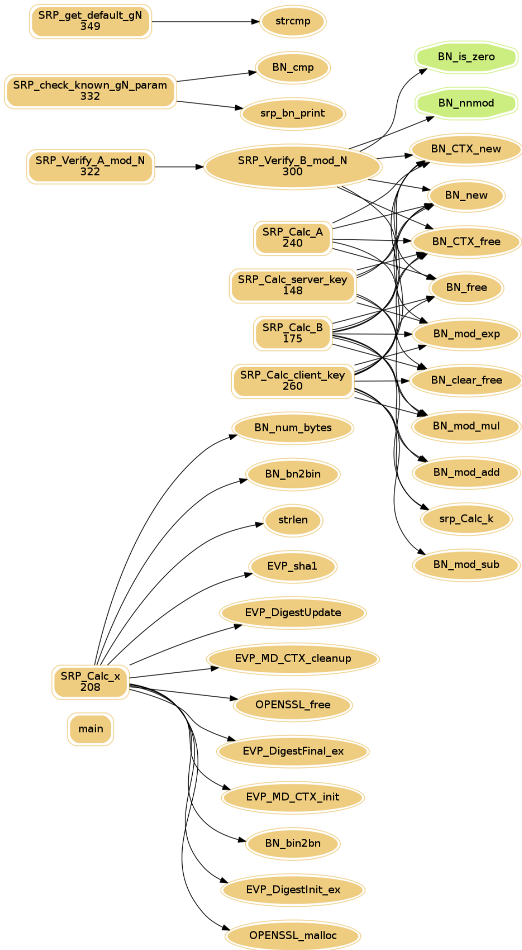

FIGURE 3.11. CVE 2014 3512(Old): Pre-Source Code Change

FIGURE 3.12. CVE 2014 3512(New): Post-Source Code Change

see whether there were key differences between control flow graphs. Figures 3.12 and 3.11 show the control flow graphs, pre-and-post, for CVE-2014-3512. The vulnerability involved multiple buffer overflows created in crypto/srp/srp lib.c. I used a diff checker to rank all the file modifications. The change between these two files was 56.13%. The repair made was a Null and value check over three variables to makes sure they were less than a given large number. The function changed, initialized, and updated the digest of a hash. Small code changes can significantly change the control flow of a program.

Most of vulnerabilities seen were DOS-based and the second most relevant vulner-

abilities were crashes due to exploits. The least common attacks were exploits based on side- channels and race conditions. After 2012, the DOS-based attacks increased significantly. There was also an increase in side-channel-based and unknown-based attacks. There were no records of side-channel based attacks prior to 2009. The only attacks sent in 2006 were DOS, buffer-overflow, and memory-based attacks. Similarities between vulnerabilities can better prepare internal developers to tackle code audits or find or repair vulnerabilities, since many of the TLS implementations, especially smaller ones, are forks from different tools. Thus, I looked at some preliminary statistics on the function calls and other static features between vulnerabilities, then went deeper by looking at all the control flow graphs in each C program. The reasoning for focusing only on the C program files was because the majority of the OpenSSL project was implemented in C and most of the critical code sections were also in C.

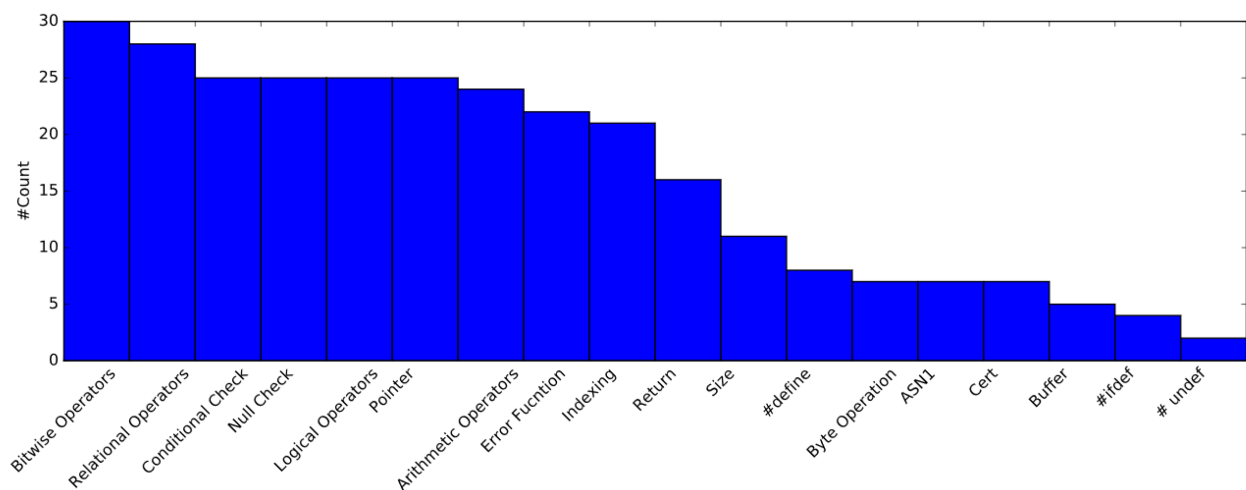

FIGURE 3.13. OpenSSL: Source Code Diff Features

I investigated the individual file changes to the whole file structure. In part of the work, I focused on the commit files, while in the other part, I focused on the complete files involved with the security vulnerabilities. The reason for focusing on a single file was that many security bugs are found in a single function or file. For the purposes of the preliminary study, I addressed each vulnerability diff file and how the files were repaired for any vulnerability in order to see whether more complexity was added to the code base.

Figures 3.8 and 3.9 show the changes in the files and how many lines were added or removed. My findings show that the functions are removed, but the number of lines of code actually increased. This was independent of the type of vulnerability.

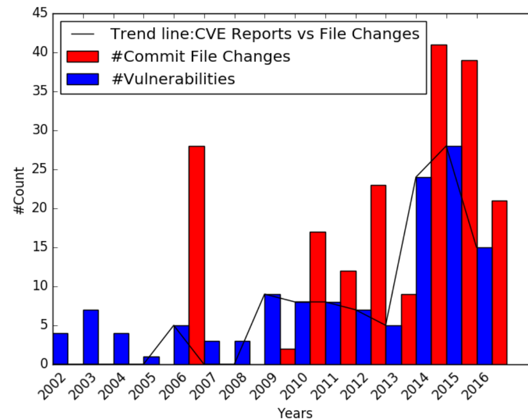

FIGURE 3.14. OpenSSL: Vulnerabilities throughout the years

Most of the features I checked are available in many languages, with the exceptions of TLS such as related features certificate check. Additionally, error function returns were also common. This tells us that error checking is of interest when considering vulnerability detection. Using a return statement causes execution to leave the current sub-routine. The less crucial features are related to certificate or ASN1 notation. This is because OpenSSL is for many operations, other than just certification and verification.

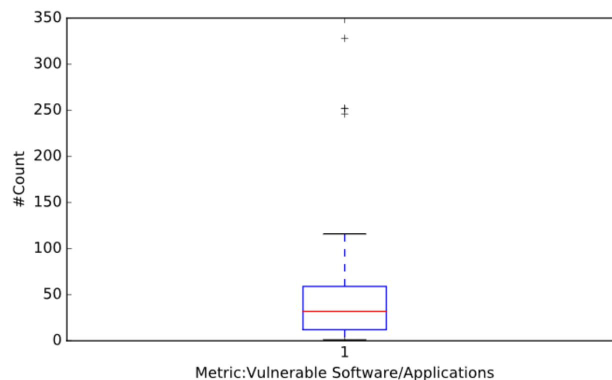

FIGURE 3.15. OpenSSL: Vulnerabilities Applications

At the preliminary stage, each code change was manually classified in terms of whether it was an isolated issue and whether the vulnerability is causally related to error handling. I found that the majority of the vulnerabilities repaired were isolated to either one function or file. However, in the commit logs, I identified multiple files that are not related to security. Major vulnerabilities can be created with small code sections. Less than 20% of all the patches to the source code base to fix vulnerabilities require changes in more than one file. A good percentage of these commits make changes to other files for documentation purposes.

I classified bugs according to each vulnerability fix entry in terms of whether it was for error handling or different issues. I identified 54% handling exceptions. Exception handling refers to vulnerabilities caused by improperly checking a logical or syntactical boundary condition. Half of the vulnerabilities were related to isolated error handling. ASM or assembly instruction calls are used in repairing code. I only identified three instances that used the ASM function. Using assembly instructions is an uncommon practice for fixing vulnerabilities.

The sizable percentage of vulnerabilities seen in OpenSSL was based on exception handling and most of the vulnerabilities involved in a single file. This supports the findings of Zaman et al. that vulnerabilities are found in fewer files compared to other bugs [53]. There was enough evidence to show that developers should pay close attention to exception handling when developing SSL and TLS packages.

3.6. Experiment Setup

I developed both a workflow and a framework for analyzing the programs. A workflow, as defined in this research, is the process during the analysis, and a framework shows the overall flow associated with the workflow. Showing both is important, as the workflow shows the actual execution process, while the framework shows how the method works overall. Figure 3.17, shows the workflow used for the experiments. The approach is merged into the build process provided by Samate. Thus, I used Wllvm in order to backtrack to build the LLVM-IR. Wllvm is meant for building llvm bitcode files using source files. Wllvm stores information on the locations so the utility can be used to generate the bitcode file. After

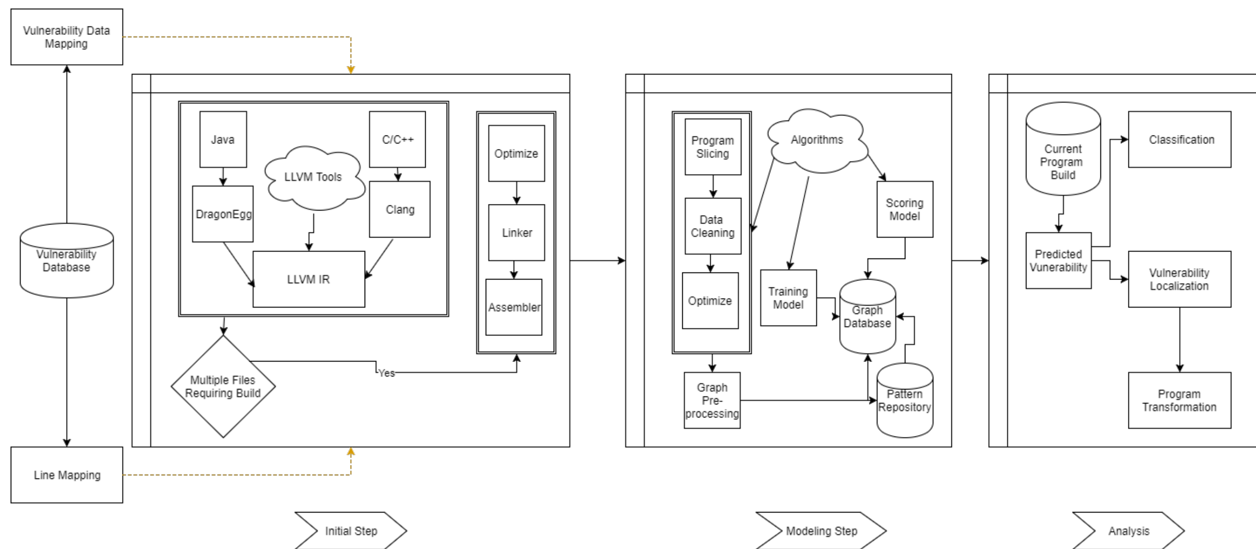

FIGURE 3.16. Framework

the compiler is replaced to use Wllvm and once the file built, the bc can be extracted. LLVM gives support to take the bc to llvm-ir form. I also used BEAR for backup using the build process. The main function of BEAR is to listen to all the calls running during the build process. BEAR runs and generates the build information. Adding BEAR into the build process increases the build time, but with a few commands, the program can be rebuilt using the same configuration presets as the first build. A dump is all the environment information that was created so the environment can remain the same as during the build process. The final item in the process is to run the make and build process. The next step is to analyze the build process, which is addressed in more detail in the framework section. Given the current technique used, if the build process fails, the program execution will end there. This is similar for many of the current static analysis tooling. If the system cannot compile, the analysis is not run. If the file can be built, there will be several artifacts: the compiled code, metadata, and the source. The metadata has information on how to find the data to link the files together, then bitcode is extracted. From the bit file, some reversing is done so it can cover the built code back to LLVM-IR. Graphs are generated using a custom llvm parser.

Figure 3.16 shows the framework used for the experiments. For the analysis, I used

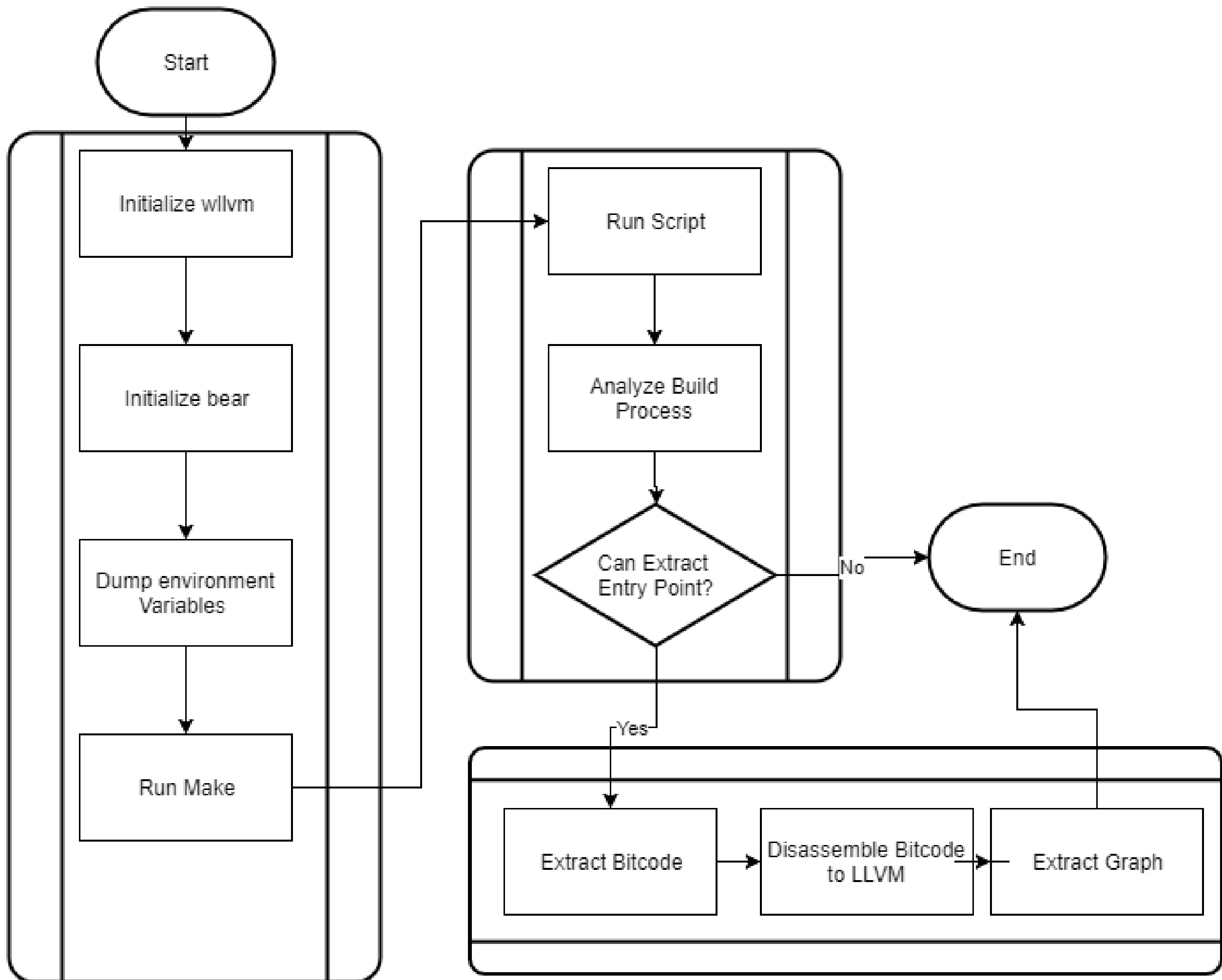

FIGURE 3.17. Workflow

graphs and patch, and captured the locations using the workflow process. The framework entails two steps: modeling and analysis. The assumption is that some vulnerability data information is available in the vulnerability database. For the purposes of this analysis, I stored the vulnerability locations in memory or the file system. For a more robust system that supports distributed computation and storage, it would be recommended to use an actual relational or graph database. I used a non-relational database because dot language converts easily to a JSON or XML file. If multiple files need to be built, graphs are linked together during analysis. The modeling stage shows the different options available. Once I generated the graphs, the whole program could be seen in different modules. Depending on the algorithms and vulnerabilities, I performed data cleansing, optimization, or program

Nodes

Node	Color
Function	Light Blue
Basic Block	Green
Instruction	Grey

Edges - Basic

Call Graph	Blue
Domain Transform(Function to BB)	Red
Control Flow Graph	Green
Domain Transform(BB to Instruction)	Pink
Instruction Control Flow	Grey

Edges - Adv

Data Dependence	Orange
External Dependence	Purple
Control Dependence	Navy

TABLE 3.3. Graph Properties

slicing. For program slicing, I extracted functions. To isolate vulnerabilities, I focused on the function level, rather than the basic block level during development. Functions are usually a common marker to show issues. After isolating the code, I continued graph pre-processing using a pattern repository. For the purposes of this study, I used custom heuristics. The training model is when different models are used in conjunction with the pattern repository. The final part is doing the scoring modeling. This involves using known vulnerabilities to assess performance. For the analysis process, I used predicted vulnerabilities to create classifications.

In the graphical representation, the goal is to gain the most meaning while minimizing the trade-off involving representation. I followed the known patterns with LLVM, which have a node structure such that the hierarchy is $Module(M) ==> Function(F) ==>$

$BasicBlock(BB) ==> Instruction(I)$. This is the same for LLVM IR. The coloring in the language was used to show types of nodes and transfers between domains. Table 3.3 shows the types of nodes and edges used in the graph representation. Nodes are either functions, basic blocks, or instructions. Each graph denotes one module. The reasoning for separating the graphs was for the benefit of reducing the overall size of the graph while still allowing for the exploration of linkage between modules. Given the configuration, exploration between graphs would have been implemented on the developers' end. There are two types of edges. One set of edges shows the change in domains and the connections between the instructions. This allows an algorithm to change between domains over a given edge of the proper domain. The advance edges show more complex relationships. I computed the control and data dependency. The control flow graph is captured as a normal edge. With so much information in the graph, the results of the analysis extends to any graphical representation. Inside each node, LLVM uses addresses corresponding to memory locations during the compile phase. While these addresses change depending on the creation of any graph, there are several advantages to using memory locations rather than strings. Using the memory locations gives a user the ability to do a regex (regular expression matching) to find all places that access that memory location with a direct cost associated with it.

Most graphical representations do not give dependences backed by internal mapping between instructions if they use the LLVM IR form. However, with external mappings, there are dependencies between functions. Thus, a very wide range of graphical representations can be inferred using the information provided. This allows for flexibility of extending the representations using different methods.

3.7. Learning Models

I built rules and used learning algorithms for the analysis of the different datasets. Traditional programming involves data and a program that passes information to the computer to generate output. Machine learning takes in data and output to then feed that information to a computer to generate a program. The program generated is the model that is re-used in many cases. Machine learning is for the purpose of information extrac-

tion, debugging, and security. Like graph problems, machine learning techniques rely on a significant number of program representations. Machine learning includes decision trees, support vector machines, and even building sets of rules. I used several learning techniques throughout this work based on the properties on the node frequency and inverse group frequency. For the purposes of this analysis, the group is the program's functions. There are also several types of learning techniques. In this study, my focus was on supervised learning, which is inductive learning. This is when the training data has information about the desired output. In unsupervised learning, the training data does not have information on the desired output. While there are also categories of semi-supervised learning, reinforcement learning, and neural networks, this work focuses on predictions using graphical representation. I used different techniques to evaluate the approach, including assessing the precision, recall, and f1-scores. I did not include other evaluated metrics, such as squared error, involving performance. Tan et al. stated that precision and recall as precision determines the fraction of records that turn out to be positive in the group the classifier has declared as a positive class. The higher the precision is, the lower the number of false positive errors committed by the classifier. Recall measures the fraction of positive examples correctly predicted by the classifier. Classifiers with large recall have very few positive examples misclassified as the negative class. The value of recall is equivalent to the true positive rate [46].

3.8. Extracting Meaning from the Core Graph

The core graph I used to analyze programs is a directed graph G that is a tuple with a vertex set $V(G)$, and an edge set $E(G)$. The relationship between vertices is a function of an assigned edge as an ordered pair of vertices. There are endpoints to and from locations that use nodes and vertices interchangeably. There is no simple graph, because there are many loops and several edges between vertices. Some of the feature extraction techniques I used are listed in Table 3.4.

To analyze the data, I processed the data and converted it into features, which I used for my machine learning models as a feature space. Instructions make up a program based on features of the number of nodes. As a working example, C-C196A-OSSL-05-ST04-

Base Op (Stemming and Lemmatization)	Second Level Analysis	Modifiers
Instruction Type Op(op)	Node Frequency	TF-IDF Variation
Instruction Type Op1(op1)	Edge Frequency	
Instruction Type Opn(opn)	Node Edge Frequency	

TABLE 3.4. Feature Extraction Techniques

DT03-DF13-CF22-01 01 is a work test case that shows the different data pre-processing techniques. This function generates 1,223 nodes with 1,847 edges; the average in and out degree is around 1.5. From a general perspective, given that each instruction references different registers, each one is unique and will not give much meaning for analysis in that form. The instruction "%7 = load i8, i8* %incdec.ptr, align 1, !dbg !277, !tbaa !235" is generated by the program. The instruction is highlighted in grey in my dot form. The node is labeled "load: incdec.ptr" and during the analysis phase, the compiler used the address 0x17c3388 to point to the instruction. To handle the variation between instructions without removing too much data, the instruction was based on operand type. In the case of that instruction, it would be load: 15. This means that this instruction is a store and location type 15. There are 55 functions: Op, Op1, and Opn. Op defines both the operator and operand in an array where the first index is the operator and all other indexes are operands. Op references to only using the op as the base feature such a call or br. Op1 means that the process used the op and the first item in the operation. Opn means to take all the operator and all value information after the op. Changing between op and opn affects the overall performance of the machine learning models. When I use Opn, the number of features in the feature space would be far larger than the feature when just using the Op or Op1. Reducing and changing the op type is known as stemming and lemmatization. This approach omits some information to gain a better analysis. If everything is included in the instruction, the feature space becomes too large. Therefore, I used ordering to figure out how to reduce the instruction type. Since this is a tiny language, reductions can increase significantly. The amount of instruction is several times larger than the amount of the basic blocks. When

looking at the instructions, there several types of call operators. However, the loads also occurred frequently throughout the module. If those points are taken based on the node and edge counts and put on a graph, the vector where each item is located is an axis. With a small dataset, it is possible to analyze the data with only the count vectorizers.

There are two problems I needed to address after count vectorization: the first relates to the graph if any regression is created, and second is having features found far out from the other nodes (i.e., the data representation does not cluster well). When key features are too far apart, the performance of calculations is hurt by under or overfitting. However, count vectorizers still have some meaning. If the count vectorizers are thought of as a feature space, Euclidean distance can be used to calculate the Euclidean norm. I examined the norm between vectors and reported the difference. While this research included the Hughes effect, also called the curse of dimensionality, this can be addressed with a more extensive dataset (i.e., training data). I treated frequent nodes as stop words in this research: for example, term references to the nodes in the core property graph and how they stand for the data. These can be the node and the edge properties. The document is the delimiter. The Property Link Frequency is the frequency of connections to each different node. Some connections provide this depending on the op level set at the first start. Increasing the degree of the feature space involved with the instructions means increasing the number of individual property instructions. As a program increases in size, the number of types of instructions will normalize out to the point where using opn becomes a practical feature space. The vertices edge function inverse vertices edge function is a novel approach borrowed from machine learning TF-IDF. In many ways, TF-IDF is an extension of the bag of words where types or words are counted. Since this technique is borrowed from text classification, I viewed the texts as items and the documents as collections. These items can be anything from types of instructions to names of the building. The goal is to get the data into vectors, so the data can be analyzed. When looking at figures discussed previously, many data points occur often, which will have adverse effects on the dataset and analysis. To address these concerns, I encoded the frequencies of words to improve the overall performance. Using TF-IDF, I

transformed the data into a normalized term frequency weight by applying a normalization. Shown below is the TF-IDF algorithm:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{1}$$

$$df_i = \log \frac{|D|}{|d : t_i \in d|} \tag{2}$$

3.9. Addressing the Core Graphs Features and Classes Problem

Converting core graph to features is a hard problem. As a core graph, the data is in a rich form, which saves a lot of information about the program's problems. However, in graph form, the graphs are not easily converted for different problems. Further, given that not all problems using the graph need all properties, it is necessary to develop a systematic way to transform the graphs into a machine learnable space. Thus, I developed several variations of the core graph to use for analysis of different vulnerabilities. The second problem is that, although the core graph has information, it is unknown specifically what information is important. While it is certain that there are vulnerabilities in the data, the challenge is deciding what to look for in the applications. A naive approach would be looking for vulnerabilities vs. non-vulnerable code. However, this approach generalizes all vulnerabilities. For example, the feature that generates an injection problem is different from the feature that causes issues with number handling. These problems create several variations of datasets to analyze, which will yield different performances for models. Unlike traditional learning problems, there are numerous feature classifications for vulnerabilities. To address the first problem, I generated several variations of features. By changing the nodes in the graphs to create a feature space, this creates a permutation in the whole graph that will affect the performance of different models and rules. I created several variations of a graph node: Op, Op1 and Opn. Operands are a key part of a program if the focus is to look only at the operands and ignore the other information, such as the return location. At this point, LLVM has done some dead global elimination, inlining, and memory promotions.

The feature would capture information about the loads, stores, call instances, and return calls. For the op flag, I created a frequency count for all the operations. Op1 implies to take the operands and first variables. Rather than using the first variable, the variable types were used so instructions can be generalizable. Opn means including the instruction with all the instruction property data. By using different vectors, this will increase the feature space. The next consideration is whether to encode the nodes, edges, or nodes and edges into the features. Again, this affects the feature space size, but can improve the performance. I denoted this information by using op, edge and op edge. Op implies only encoding the nodes in the features space. Edge implies only encoding the edge and not the ops into the feature size. Finally, op edge means to encode the features space the operands and the relationship into the feature space. Edge with Opn would have the largest feature space, compared to the other permutations. Many vulnerability databases have their own categories. For the scope of this dissertation, I have categorized these into five groups. The five groups are: stone cwe mapping c, cwe mapping, cas cwe mapping, cwe id, and cwe type id. These are different class groups. On a larger scale, there are vulnerabilities, groups, and IDs for specific vulnerabilities. This is a common hierarchical labeling of most vulnerability taxonomies. CWEs are common software security weaknesses. I removed test cases if I did not find more than 20 records for given vulnerability classes in stonesoup because having very few vulnerabilities would make it hard for users to infer the correct properties. From this information, I used 90 datasets for the NIST analysis. For each testing dataset, the number of classes ranged by 77 points. I used the min-max between 19 and 96 in terms of classes to identify types of vulnerabilities and whether the function was identified as safe.

3.10. Other Approaches: Bug Prediction

I compared several bug prediction approaches in the results analysis section. These include algorithms by Wang, Halstead, and Nguyen [7, 10, 35, 37, 47, 48]. I found Halstead features to be useful in creating defect predictors based on static attributes by looking at the complexity of the program. I considered the complexity of the code after it is converted into the intermediate representation and has captured several additional dependencies involved

with the application in the graph. The features I used to create the program complexity features were:

- n_1 = the number of distinct operators
- n_2 =the number of distinct operands
- N_1 =the total number of operators
- N_2 =the total number of operands

The next group of listed equations shows the program complexity feature derived from core features. These features represent the program complexity shown in the application, but do not address the relationship between connected instructions. The program length is the total number of operator and operands used in the applications. Halstead does not use edge information to determine the complexity. The vocabulary size is defined as number of unique operands and operators. The volume is the information contained in the program.

- Program vocabulary: $n = n_1 + n_2$
- Program length: $N = N_1 + N_2$
- Calculated Program Length: $\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- Volume: $V = N * \log_2 n$
- Difficulty: $D = \frac{n_1}{2} * \frac{N_2}{n_2}$
- Effort: $E = D * V$

Nguyen created features that included data flow [40]. He defined mode and edge, as well as component features. However, for the purposes of this study, I focused on the member node attributes. All of the member node attributes are self-explanatory, other than the McCabe complexity. McCabe introduced cyclometric complexity, which measures the complexity of the code. The McCabe complexity (M) shows the increase in changeability. Once the program graph is created, the complexity M is defined as $M = E - N + 2P$. E is the number of edges, N is the nodes of nodes, and P is the number of connected components. In a small program, connected components are 1.

- L =line of code
- LoC =Lines of comments

- LB = blank lines
- V = McCabe complexity
- P = the number of parameters in the method signature
- E = the number of return points in the method

The final feature space is built using features introduced by Wang et al. [47]. The feature space addressed by Wang et al. is similar to Nguyen's. Wang et al. added fan-out and fade-in. Fan out is the number of other classes referenced by the class, while fan in is the number of other classes that are referencing the class under investigation. Wang et al. also used McCabe's complexity in their feature space. Features selected for this study from Wang et al. are:

- L = line of code
- V = McCabe complexity
- MFC = Method Function Count
- FO = Fan-out
- FI = Fan-in
- MR = Method References

CHAPTER 4

RESULTS

This section is divided into several parts. The first section covers the use of NIST OpenSSL test cases to evaluate different approaches. For each application in NIST dataset, I performed an analysis on each vulnerability grouping. I then grouped all NIST datasets together to form one corpus for testing. The final section goes into OpenSSL testing. Given the size of the corpus when dealing with the NIST dataset, I focused on the fast models because some models, such as support vector machines(SVM), have a complexity approximation of $O(sample^2 * features)$ based on the implementation used. The amount of storage increases for the training vectors. Thus, the results are limited to the fast models in the datasets for performance reasons.

4.1. Analysis of the Data

I created several models to analyze the different performances. Analysis used included decision trees(DT), stochastic gradient descent (SDG) classifiers, random forest classifiers, and a few others. I selected classifiers based on the speed at which they could handle large computation and their ability to handle large datasets. Rather than reporting on all the information collected, the focus was primarily on the top performing models from the datasets in the first few sections and summarizing the results in the final sections. For each section in the first part, the dataset looked at the models that had the most test cases' classes classified correctly with a high accuracy, and used a two-fold cross validation. The following sections address the NIST OpenSSL dataset to evaluate the performance of the different models. I focused on the decision tree, random forest, and SGD classifiers due to the computation speed compared to more complex algorithms. The key benefit of using stochastic gradient descent algorithms for the analysis is the efficiency and ease of implementation. The drawback is that these are sensitive to feature scaling, which can have adverse effects on performance. I used the term "performance" in this document to imply a relationship to the precision, recall, accuracy, or f1-scores. Performance can refer to one or

all metrics. SGD classifiers create different loss and penalty functions for each class. Decision trees are a very old modeling technique, but are an effective approach, with the benefit of being fast compared to other algorithms. Decision trees are a non-parametric, supervised learning method that is easy to understand. Also, there is a logarithmic cost for building the trees, which means they can be constructed fast. The key drawback to decision tress is that the algorithm can be greedy and suffer from overfitting. Random forest is an ensemble classifier based on decision trees, with the aim of improving predictive accuracy and control over-fitting. Random forest works by randomly selecting subsets of an independent feature in the feature space. Then, for each one, they build decision trees based on created leafs. The collection of decision trees is a random forest. Each decision can perform classification; however, in a random forest, multiple trees are used to decide based on the majority. I ran over 400 tests against different model configurations using different feature extraction and creation techniques. The goal was to explore the limitations and advantages of different models and model configurations without losing any testing information. Even within these tests, I used a grid search algorithm to find the best model with different configurations. The next sections include a discussion of the analysis of the best models, given that I used a grid search for each graph feature extraction algorithm.

4.2. NIST SAMATE

4.2.1. DT OpenSSL NIST: Simple Classification Using Edges with OpN

Decision trees are powerful when it comes to a multi-output problem, because researchers can generate and view the tree and decision boundaries. In general, decision trees are also a great starting point for any learning problem. I compared results from the decision tree analysis using Gini and entropy algorithms. The significant difference is that Gini does not need the algorithm to compute the logarithmic functions, which are marginally computationally intensive. Gini addresses misclassification while entropy is an exploratory analysis of the data. I tested different models with different tfidf configurations. Out of all the feature extraction techniques used, I found that simple classification based on vulnerable vs. not vulnerable worked best with the highest average precision. For the best performing model,

using only the edges with each type operating worked well. For the other testing, DT using simple classification was able to achieve at or above 90% for precision, recall, and f1-score. When I attempted to classify vulnerable functions based on CWEs, the performance metric dropped significantly.

Figure 4.1 shows a confusion matrix involving the best decision tree classifier for the NIST OpenSSL dataset. There were only five functions misclassified. Four classified vulnerable functions are not vulnerable. Overall, the model performed well. When only looking at the confusion matrix, using only a decision tree can be effective for deciding vulnerabilities in the OpenSSL dataset provided by the NIST SAMATE dataset. The results show that the best configuration was to use only the edge information and all the operand information. The best model based on accuracy was Gini while using tfidf; however, the increase in accuracy was less than 0.1%. For the data obtained using only the OpenSSL dataset, the model achieved around 98% correct detection when using a decision tree.

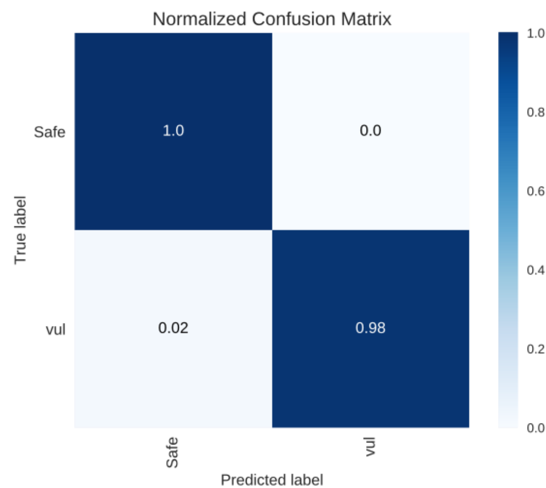

Figure 4.1. DT OpenSSL NIST Confusion Matrix: Simple Classification Using Edges with OpN

Table 4.1 shows the precision, recall, f1-score, and support I used to make the calculations. I found the decision tree to be effective in determining precision, recall, and f1-score. The only metric that shows a slight decrease in performance is recall, with only a marginal

	precision	recall	f1-score	support
Safe	1.00	1.00	1.00	11523
vul	0.99	0.98	0.99	178
avg / total	1.00	1.00	1.00	11701

TABLE 4.1. DT OpenSSL NIST Report: Simple Classification Using Edges with OpN

change. Using a decision tree, I identified a performance of 98% regarding precision, recall, and f1-score. I ran 54 tests using different configurations and extraction techniques for the graphic representation of the dataset.

4.2.2. Random Forest Classifier OpenSSL NIST: Simple Classification Using OpEdges with Op

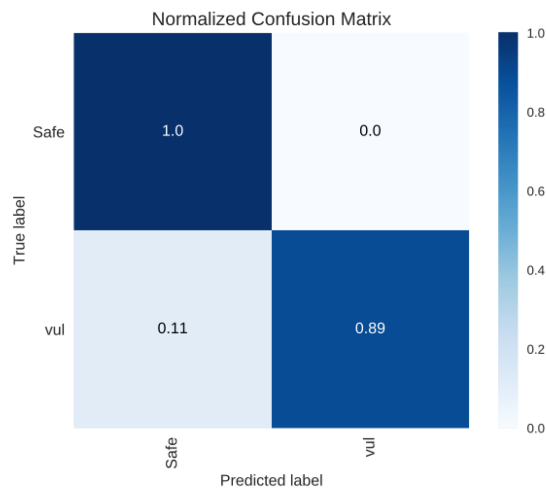

FIGURE 4.2. RandomForest Classifier OpenSSL NIST Confusion Matrix: Simple Classification Using OpEdges with Op

After my decision tree analysis of SAMATE OpenSSL, further research is needed

	precision	recall	f1-score	support
Safe	1.00	1.00	1.00	11545
vul	0.99	0.89	0.94	156
avg / total	1.00	1.00	1.00	11701

TABLE 4.2. RandomForest Classifier OpenSSL NIST Report: Simple Classification Using OpEdges with Op

to explore different models and provide a better understanding of extraction methods and model configurations. Even though I used cross-validation with training and testing split, it remains important to see whether the decision tree will scale or fail when comparing the model against to other learning algorithms and configurations. Random Forest Classifiers was a good natural progress, given the success of my results with the decision trees. I used the configuration and knowledge of the decisions to reconstruct, extend and test it against an ensemble classifier. When looking at only the confusion matrix, the random forest exhibited a decrease in performance when detecting vulnerabilities. Nineteen functions are misclassified. However, while using a random forest, I was able to detect many vulnerabilities as well as classify whether or not they were safe. That brings the best classification for vulnerable functions down by 1% compared to using the best decision trees model with features extraction using edge Opn. The best model uses opedge op information, compared to using the edge information as in the best decision tree. This information is shown in Figure 4.2.

Table 4.2 shows the overall performance with precision, recall, and f1-score. The recall decreased, which in turn decreased the f1-score. This constitutes approximately a 10% decrease in performance compared to just using a decision tree for recall. Looking at the best configuration, entropy and Gini had a negligible effect on the performance. The number of estimators was important in improving accuracy. In the optimized model for accuracy,

	precision	recall	f1-score	support
Safe	1.00	1.00	1.00	11523
vul	0.97	0.83	0.89	178
avg / total	1.00	1.00	1.00	11701

TABLE 4.3. SGD Classifier OpenSSL NIST Report: Simple Classification Using Edge with OpN

99.8% was the best accuracy, which had five estimators. The results were limited to two and five estimators. Tfidf was not a commonality in increasing accuracy of the model compared to the number of estimators. The criteria, such as normalization techniques for the tfidf, did not have much of an effect either, given that tfidf was not a commonality.

4.2.3. SGD Classifier OpenSSL NIST: Simple Classification Using Edge with OpN

The final model I tested against the OpenSSL SAMATE dataset was a stochastic gradient descent classifier. My intention in this was to get away from using only tree-based models and investigate a different algorithm with fast computation. The speed comes from the way SGD solves a problem as a minimization problem rather than trying to solve as a convex optimization problem.

Figure 4.3 shows that, while the overall performance was good for the tuned stochastic model, it's precision was not as good as the best DT and random forest classifiers. There was a 1% decrease in precision compared to the best of random forest approaches. The recall also decreased to 83%. The best performing model was able to detect many of the vulnerabilities, but with some error.

Figure 4.3 shows the confusion matrix. At this point, there was an increase in the number of false negatives and false positives. The confusion matrix shows around 17% percent of misclassifications involved with using the best SGD classifier. Out of the three

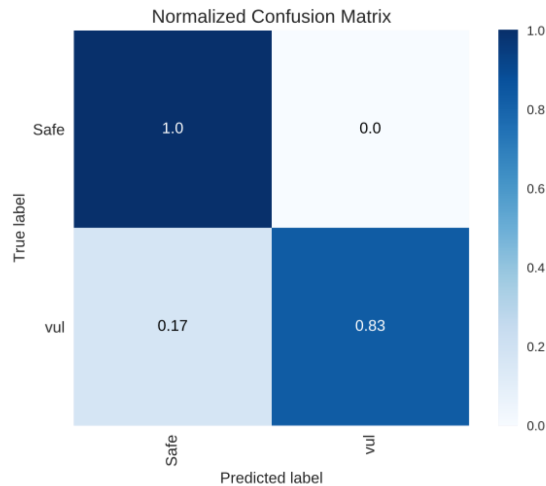

FIGURE 4.3. SGD Classifier OpenSSL NIST Confusion Matrix: Simple Classification Using Edge with OpN

classifiers, the SGD classifier, based on the best model on the accuracy, performed the worse compared to random forest and the decision tree. The SGD classifier was unable to detect as many vulnerable functions as the other two models.

The best configuration was to use only the edge information and edge opn. This is the same feature extraction for the best decision tree that gave high accuracy. In general, using log function showed the best accuracy compared to using tfidf, which did not have a significant impact on the accuracy of the model. I did not find tfidf or tfidf normalization techniques to be a strong factor in predicting how well the models would perform based on accuracy.

4.2.4. DT Classifier NIST: Simple Classification Using Edge with Op1

One important question is: how well does this approach scale to different vulnerabilities and applications? Thus, I combined all the data from different SAMATE datasets to see how well these approaches would perform. The next set of results shows the walk through of using DT on the NIST dataset. DT was the best overall model given my focus on OpenSSL in SAMATE. Significantly, the number of vulnerable functions increased dramatically, while

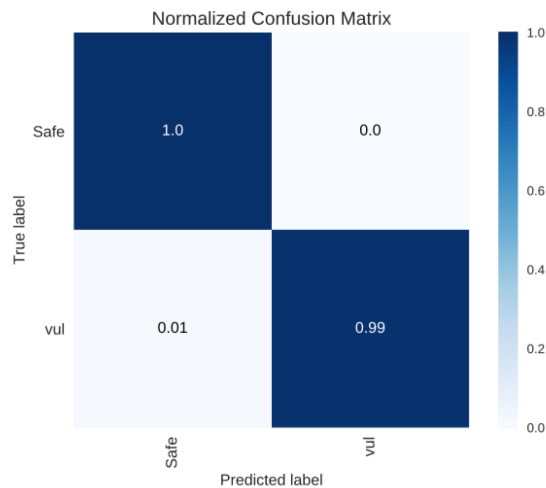

FIGURE 4.4. DT Classifier NIST Confusion Matrix: Simple Classification Using Edge with Op1

the number of vulnerable functions used for cross-validation and testing had a little less than double the amount of test cases.

In evaluating the confusion matrix for the optimized algorithm based on accuracy, Figure 4.4 show performs on par with best decision tree model tested against NIST OpenSSL. The best performing model was able to detect many of the vulnerabilities. The results showed that using any simple, vulnerable classification led to the best performance of the model than trying to identify the classification based on the CWE groups.

4.2.5. Random Forest Classifier NIST: Simple Classification Using OpEdge with Op1

The next model tested used random forest against the whole dataset. Since different classification models have distinct advantages and limitations, I considered best performing models while using a grid search. To validate the model, I identified the test data, which had 43,557 safe functions and 420 vulnerable functions. We can observe that for every 100 safe functions, there was one vulnerable function. There were significantly more vulnerable functions than non-vulnerable functions. Figure 4.4 shows that the model performed well, but not as well as the decision tree. The best data configuration involved using only two classes

	precision	recall	f1-score	support
Safe	1.00	1.00	1.00	43557
vul	0.98	0.94	0.96	420
avg / total	1.00	1.00	1.00	43977

TABLE 4.4. Random Forest Classifier NIST Report: Simple Classification Using OpEdge with Op1

and required using operands and edges, but only limiting the information to the operand and only one vector type. This configuration is different from the best NIST OpenSSL model using the random forest classifier. For analysis, I used varying sizes of trees, ranging from two to five branches. I also used different criteria, including tfidf online and different tfidif normalization. Normalization had a little effect on the overall performance. There was a notable increase in accuracy around 0.003 when the number of trees were increased. Also, the configuration using entropy exhibited better performance than Gini. In all cases, Gini performed better compared to using the different random forest configuration.

4.2.6. SGD Classifier NIST: Simple Classification Using Edge with Op1

The SGD classifier represents linear classifiers based on using stochastic gradient descent training. The loss is estimated based on the samples at the time the model is updated. Figure 4.5 shows the confusion matrix. I chose to employ the confusion matrix throughout the results section because it is a simple but effective way to show the overall performance of the model, and is intuitive for anyone to understand. The best SGD performed worse than the DT and the random forest models. The SGD model misclassified 137 functions. It misclassified 10 vulnerable functions as non-vulnerable, while misclassifying 127 that were not vulnerable as vulnerable. This means that 70% of the vulnerable functions are correctly classified. While the SGD model is a good model, it does not perform as well as the other

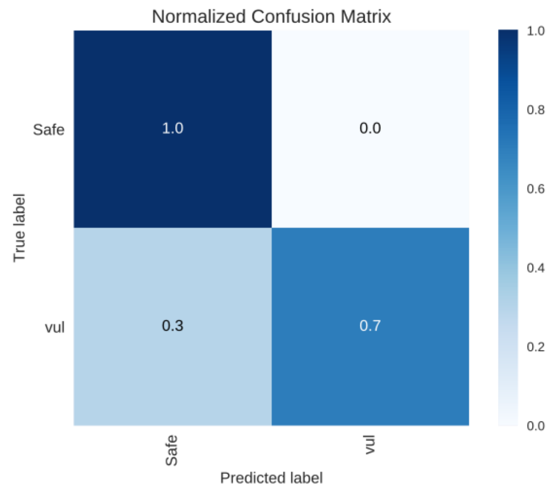

FIGURE 4.5. SGD Classifier NIST Confusion Matrix: Simple Classification Using Edge with Op1

two: decision tree and random forest.

Table 4.5 shows the metrics involved with the model testing data evaluation. Further, it shows that recall is affected the most, compared to precision for a same tuned model. This means that it classifies the wrong classes as a different label, as seen in the confusion matrix. The recall for this model was at 70% for detecting vulnerable functions. However, the precision was at 97% for detecting vulnerable functions. The precision, recall, and f1-score were high when determining whether a function was safe.

Figure 4.6 shows the roc curve for the SGD model: 0 denotes safe while 1 denotes vulnerable (vul). Interestingly, the area of the curve is the same for both the vulnerable and safe functions. However, there is sharp increase in the area under the curve compared to using vulnerable functions. The micro curve shows good performance based on the area under the curve. Overall, the roc curve shows that performance for predicting vulnerable and non-vulnerable functions is good using the tuned SGD classifier.

	precision	recall	f1-score	support
Safe	1.00	1.00	1.00	43550
vul	0.97	0.70	0.81	427
avg / total	1.00	1.00	1.00	43977

TABLE 4.5. SGD Classifier NIST Reports: Simple Classification Using Edge With Op1

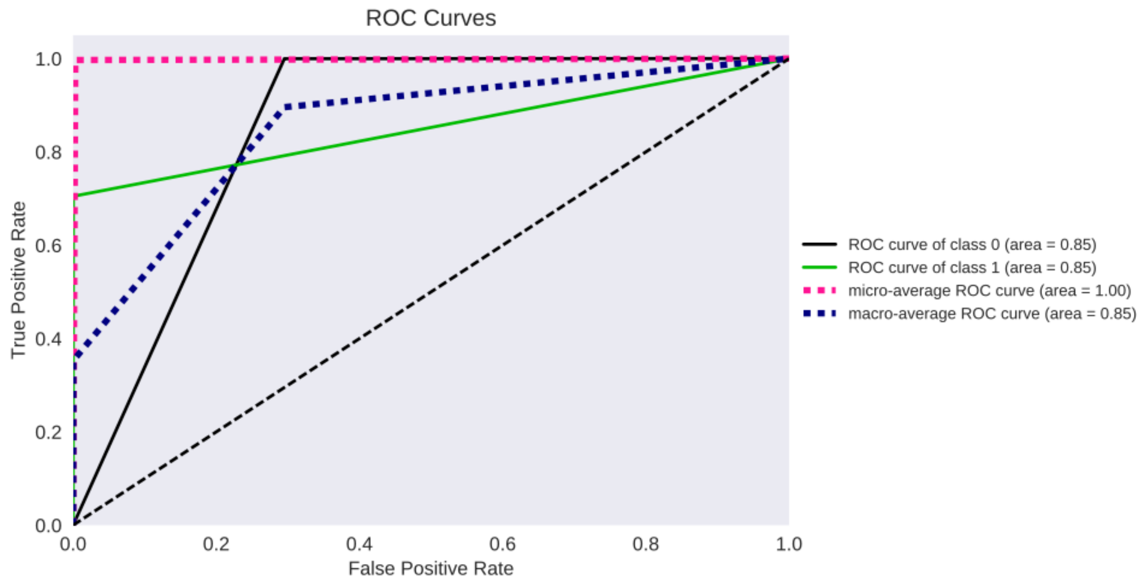

FIGURE 4.6. SGD Classifier NIST ROC: Simple Classification Using Edge with Op1

4.3. OpenSSL

For the next sections, feature extraction is kept the same to determine how well my approach worked against OpenSSL. Likewise, isolating the feature extraction methods given the environment could be tuned against the model, while still using cross-validation. This section contains results when analyzing the data extracted from OpenSSL.

4.3.1. DT Classifier: OpenSSL

Thus far, I have addressed using different learning techniques against generated data. While these functions show real vulnerable functions, they are not vulnerabilities seen in real applications. This is because it was important to consider how the process techniques would be able to perform in generated data before I tested against real applications. The application under investigation was OpenSSL, which is a very popular open source application that many larger companies use to test their tooling for vulnerable functions. Many non-crypto-companies rely on OpenSSL to provide end-to-end security for their web applications or even networked applications that run in the background. One issue that became apparent during testing was that the datasets were unbalanced. Given that the classification was unbalanced, the models were scaled to weigh in favor of the vulnerable functions. If this step were not taken, the number of false positives and false negatives would increase. As addressed in the previous sections, trying to use precise labels was a problem. Thus, my focus was on the simple labels, as they were shown, to perform better compared to trying to classify vulnerable functions manually in the last section. When looking at the confusion matrix and other metrics, my goal was to get above 50% of the number of true positives, which proved to be an area of success with different models.

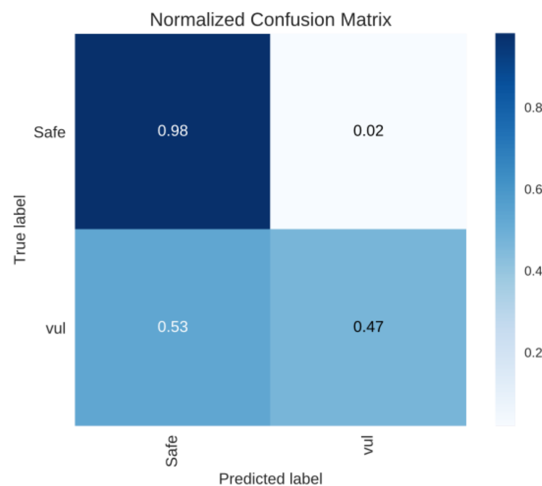

FIGURE 4.7. DT Classifier Normalize Confusion Matrix : OpenSSL

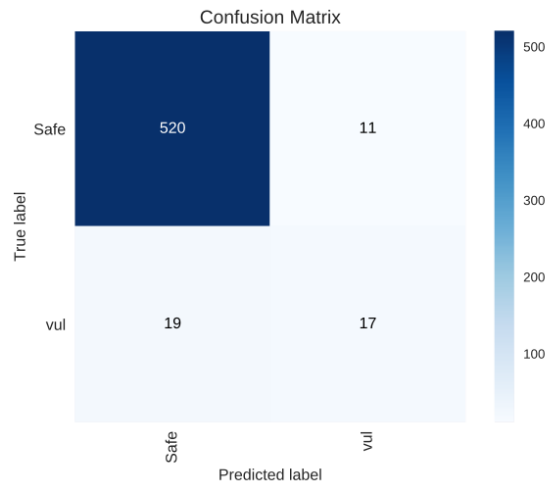

FIGURE 4.8. DT Classifier Confusion Matrix: OpenSSL

The best decision tree model performed not as well to tests ran against the SAMATE dataset. Figure 4.7 shows the percentage-based confusion matrix while Figure 4.8 shows the count. When detecting vulnerable functions, the model was only able to detect around 47% of vulnerable functions. The same model was able to detect around 98% of vulnerable functions when detecting whether a function was safe. It is important to note that a safe classification does not mean the function is safe. Rather, it means that the application did not find properties similar to the ones found in previous vulnerable functions. The model was predicting more vulnerable functions as being safe than being not vulnerable functions. However, the results show that the model performs well when it comes to predicting safe functions correctly.

Figure 4.9 shows a gains/lift chart generated by the best performing model. The line in the middle shows how well picked predictions for a given class would perform if randomly selected. Though there is no wizard line in the figure, the wizard shows the best performance. The larger the line, the better the performance of the model. The x-axis shows the percentage of the samples, while the y-axis shows the percentage of positive responses. The lift curve shows the prediction of the response model and calculates the percentage of positive responses for the percent of the functions classified correctly. The reason for the

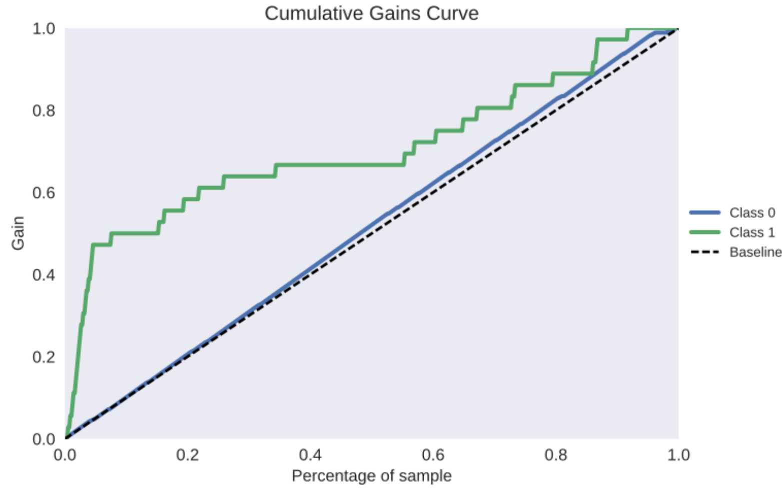

FIGURE 4.9. DT Classifier Cumulative Gain Curve: OpenSSL

	precision	recall	f1-score	support
Safe	0.96	0.98	0.97	531
vul	0.61	0.47	0.53	36
avg / total	0.94	0.95	0.94	567

TABLE 4.6. DT Classifier Report: OpenSSL

safe classification being so close to the random is because the majority of safe functions are classified as safe. The cumulative gain curve shows that using the decision tree was effective regarding the gain curve.

Figure 4.6 shows an evaluation of the model using the test data. The precision, recall, and f1-scores are still in the 90% range while vul classification is around 50% for all metrics. The ratio for vul to safe in my evaluation data was 1:14. Given the vast amount of safe functions, the low performing vul metrics had little impact on the overall performance and indicated a large number of safe functions.

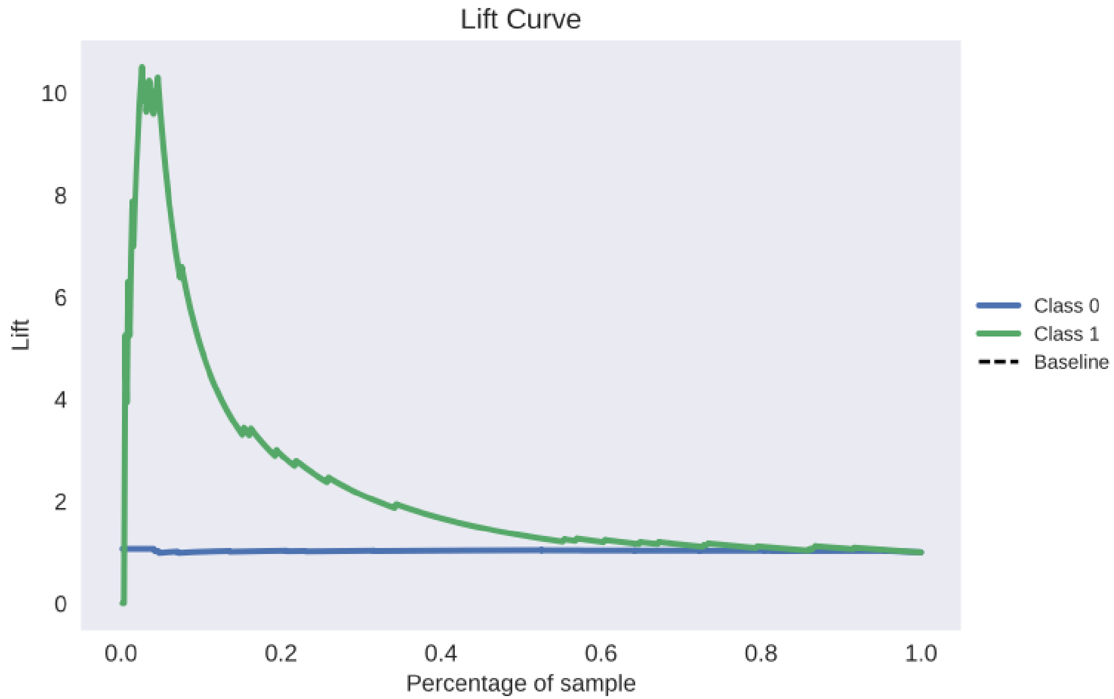

FIGURE 4.10. DT Classifier Life Curve: OpenSSL

Figure 4.10 shows a lift curve. Similar to the to gain chart, the lift curve checks the rank ordering of the probabilities. The graphs show that, at the beginning, the lift to vulnerable functions is good; however, after the sample increases, the lift decreases quickly. This shows that this model has a poor lift curve. Ideally, it would be preferable for the lift curve to remain high throughout the calculations.

Figure 4.11 shows the precision-recall curve over the percentage of sample space. The graph shows that, as sampling increases, recall and precision decrease quickly. This is especially the case when trying to determine whether a function is vulnerable or not given the current data. The graph shows a sharp drop for determining whether a function is vulnerable or not when compared to detecting when a non-vulnerable function should be safe.

In Figure 4.12, the roc curve shows the relationship between specificity, which is the

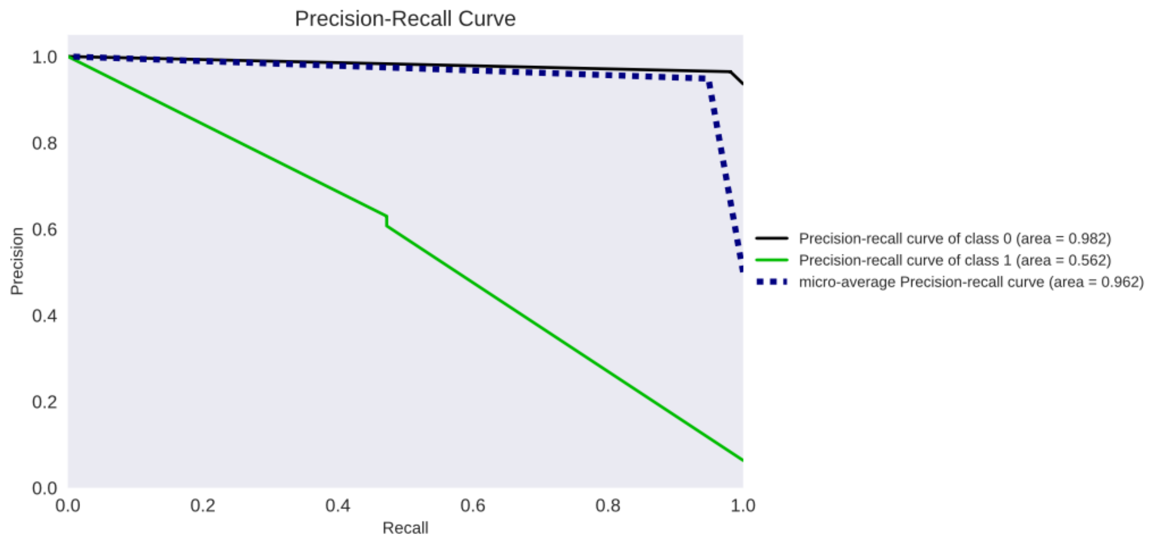

FIGURE 4.11. DT Classifier Precision Recall Chart: OpenSSL

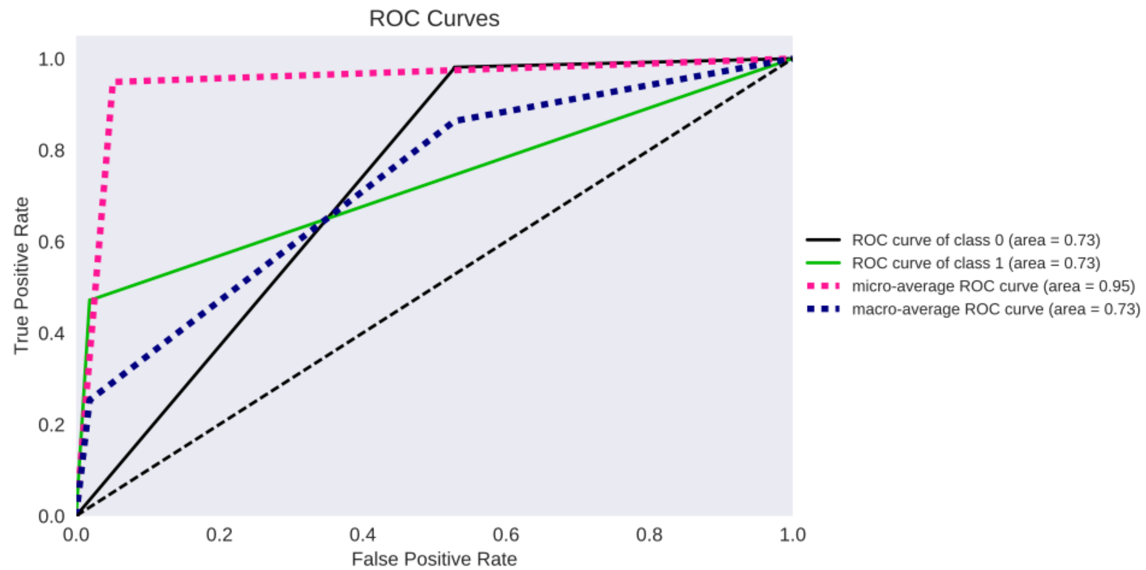

FIGURE 4.12. DT Classifier ROC Curve: OpenSSL

true negative rate, and sensitivity, which is the true positive rate. The higher the position of the curve over the middle line, the better the performance. Both classification ROCs are around 70%, which means that classification is fair. Anything above 80% would be a good

result for any classification model.

4.3.2. Random Forest Classifier: OpenSSL

I conducted the next test using a random forest classifier. The random forest classifier yielded good results against the NIST dataset. Random forest also had the added benefit of being able to find features that might be overlooked through random sampling of the feature space.

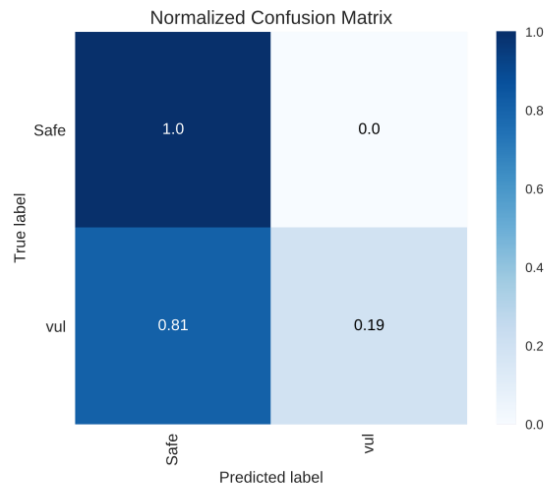

FIGURE 4.13. Random Forest Classifier Normalized Confusion Matrix: OpenSSL

Figure 4.13 shows the confusion matrix for the best random forest classifier based on accuracy. The random forest was better at detecting safe function calls in the methods tested. However, the random forest technique used was worst at detecting vul functions compared to other learning methods used for testing. For instance, 81% percent of the vulnerable functions were classified as safe. Figure 4.14 suggests that the model classified two vulnerable functions as safe. This matrix indicates that the majority of vulnerable functions are being classified as safe.

Figure 4.15 shows the cumulative gain curve for the best random forest model. The gain table also shows how much of one label will be obtained with a given percentage of the data. Both lines in Figure 4.15 are measured in percentages, which is why the safe label is

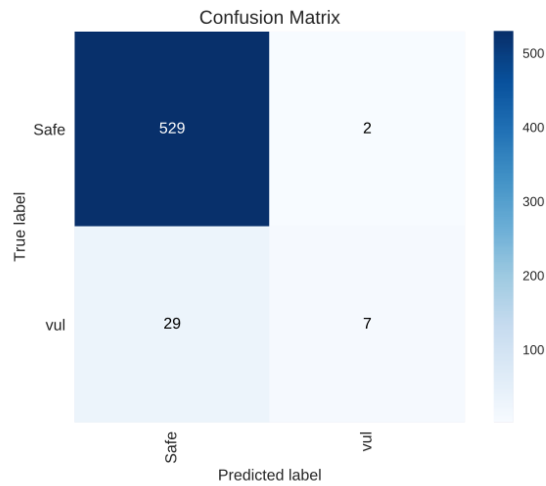

FIGURE 4.14. Random Forest Classifier Confusion Matrix: OpenSSL

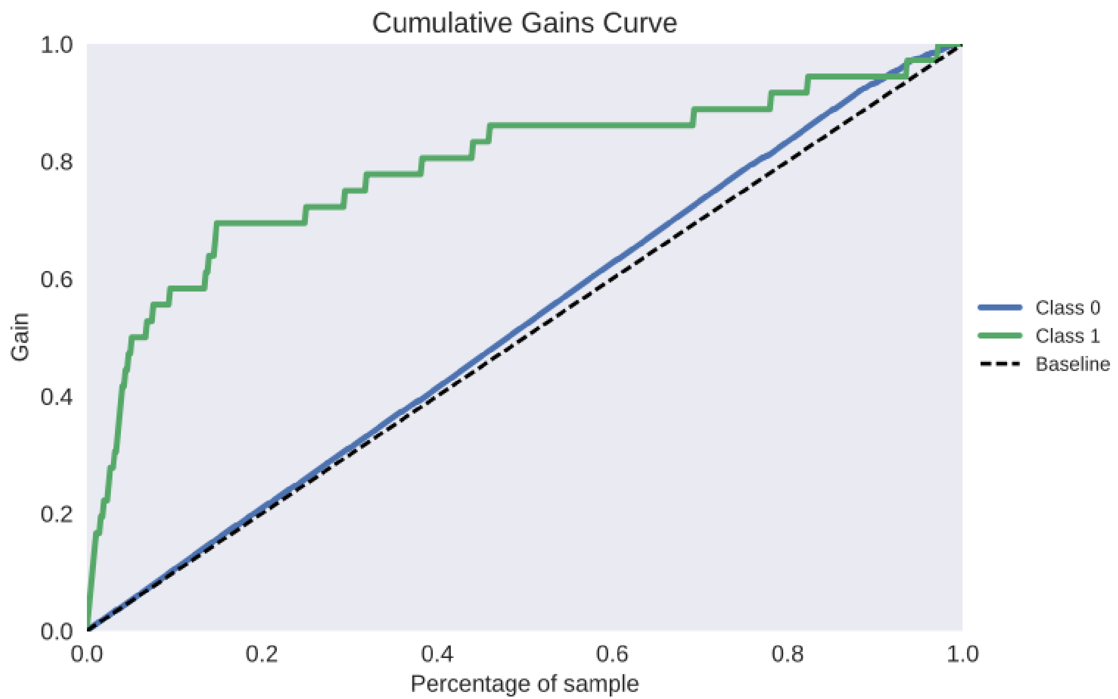

FIGURE 4.15. Random Forest Classifier Cumulative Gains: OpenSSL

	precision	recall	f1-score	support
Safe	0.95	1.00	0.97	531
vul	0.78	0.19	0.31	36
avg / total	0.94	0.95	0.93	567

TABLE 4.7. Random Forest Classifier Report: OpenSSL

the trending new middle line.

Figure 4.7 shows the confusion matrix. Classification of the safe function was high. Vul function classification was low, compared to the decision tree model tested. Having a low recall pulled the f1-score down to 31%. This f1-score is much lower than scores shown using the random forest shown in both the NIST OpenSSL and SAMATE OpenSSL. With the recall for detecting safe functions at 100%, the recall for detecting vulnerable functions is at 19%. The precision is fair at 78%. The recall for this model is about 20% lower than what was seen on the whole dataset, when running the random forest classifier.

Figure 4.16 shows the KS gains involving the random forest. Figure 4.16 shows good separation between both labels, until the 40% point. The chart shows that, early in the process, there is good separation between classes; however, as more data is collected, the model becomes poorer at determining the difference between labels. The maximum distance between the lines is the KS value. The graph shows an estimate of a KS value around 50%. The greater the KS value, the better the model might perform. As more data is consumed, the KS value goes down at a sharp rate. Early on in the data, the KS value is high compared to when more data is read into the model. The KS value also has a sharp trend around 20% of the data between both labels.

Figure 4.17 shows a lift curve, which is different from the one shown in the best decision tree model. The random forest lift curve indicates a straight down trend. However, both curves are near 0 when the data is around 40%. There are several small spikes in the lift

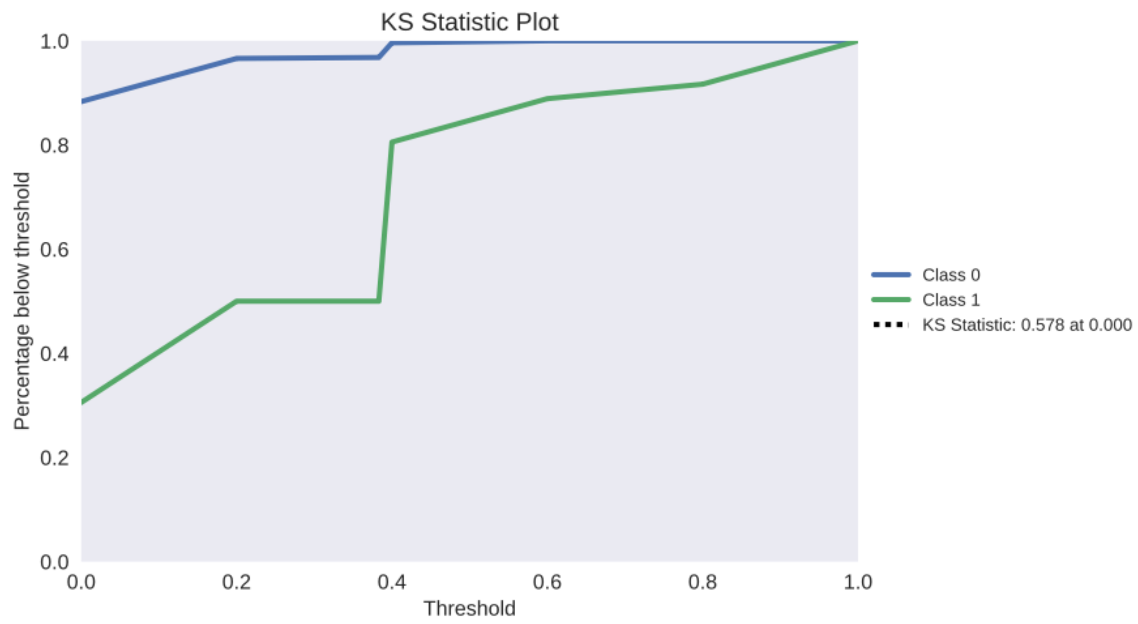

FIGURE 4.16. Random Forest Classifier KS: OpenSSL

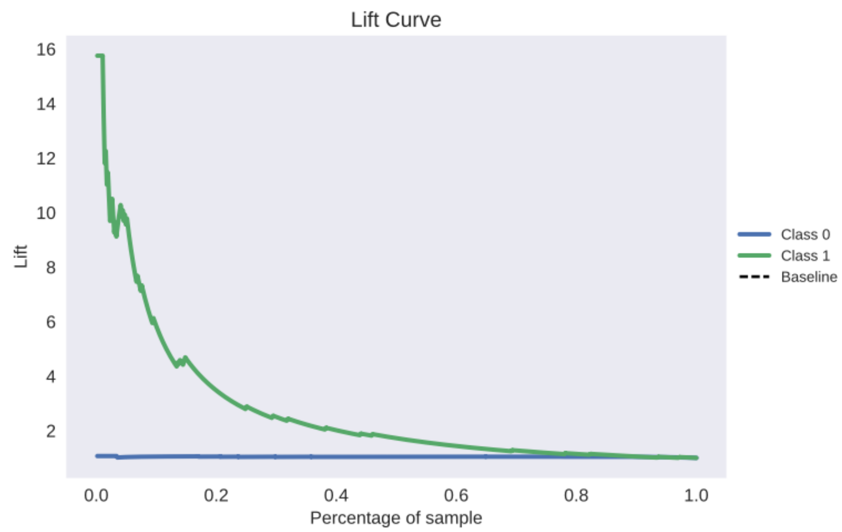

FIGURE 4.17. Random Forest Classifier Lift Curve: OpenSSL

curve as well, which show small but sudden changes with how the model segregates classes.

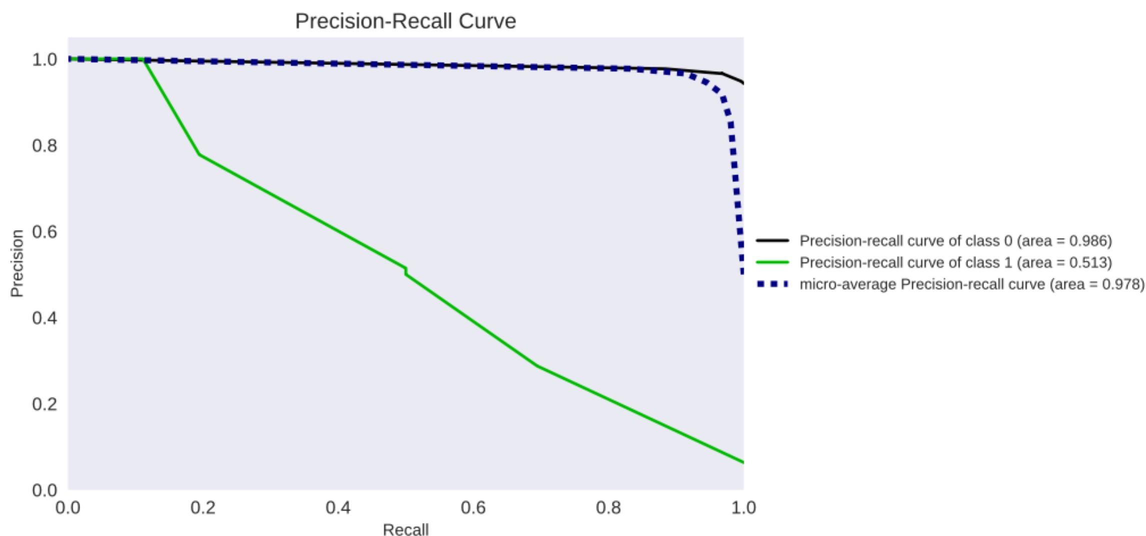

FIGURE 4.18. Random Forest Classifier Precision Recall Curve: OpenSSL

Figure 4.18 shows the precision-recall curve. The x is the recall and the y is the precision. This shows that, for the vulnerable class, the recall drops fast. The safe functions, precision, and recall stay high compared to the vulnerable class.

Figure 4.19 shows the ROC curve for the random forest classifier. The ROC of the classifications involving vulnerable functions has a higher area under the curve compared to the vul function. The area under the curve is initially higher for the vul function. However, total area under both curves is the same.

4.3.3. Linear SVC Classifier: OpenSSL

I also tested LinSVC using the OpenSSL dataset. This testing performed the best, compared to other approaches. It achieved 72% when it came to detecting vulnerabilities when looking at the recall. Figures 4.20 and Figure 4.21 show the confusion matrix. Although there is a significant decrease in precision, there is an increase in the ability to detect vulnerable functions. Further, only 16% of safe functions are classified as vulnerable. This was the highest percentage compared to the other models tested thus far using the OpenSSL

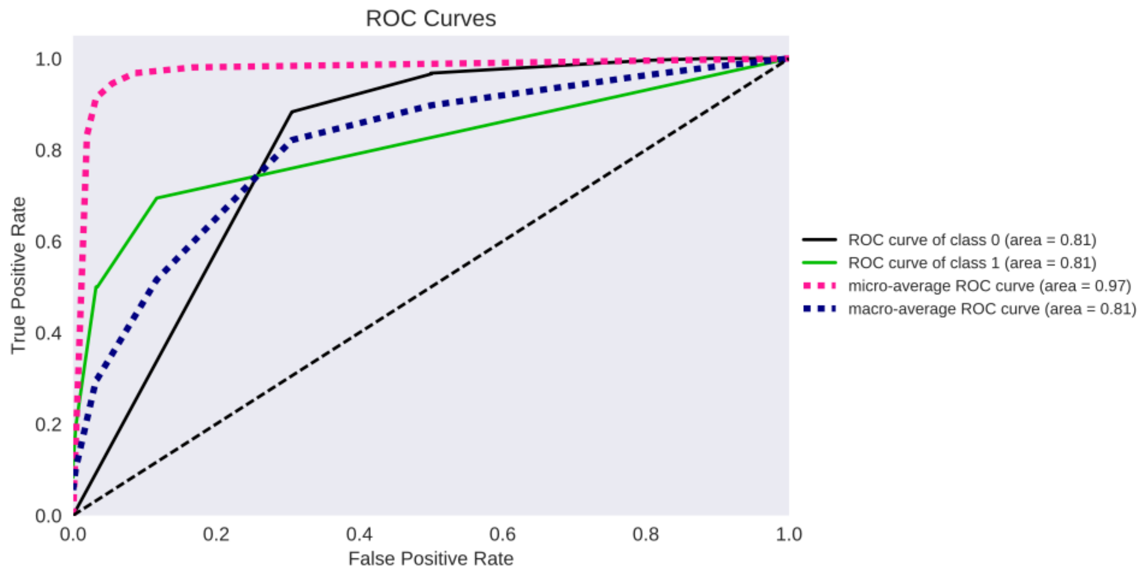

FIGURE 4.19. Random Forest Classifier ROC Curve: OpenSSL

dataset for detecting vulnerable functions. The 72% means that the best model was able to classify more vulnerable functions while not misclassifying as many as not vulnerable. In the confusion matrix, the results detected 26 out of the 36 vulnerable functions in the test data.

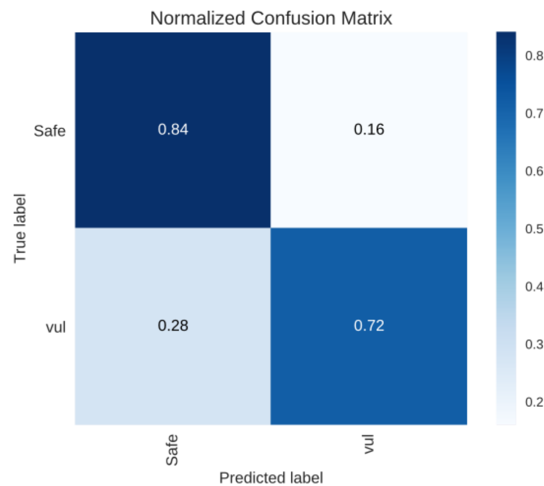

FIGURE 4.20. Linear SVC Classifier Normalize Confusion Matrix: OpenSSL

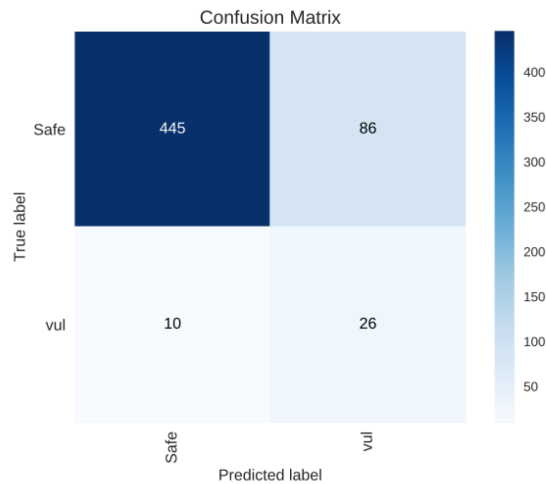

FIGURE 4.21. Linear SVC Classifier Confusion Matrix: OpenSSL

	precision	recall	f1-score	support
Safe	0.98	0.84	0.90	531
vul	0.23	0.72	0.35	36
avg / total	0.93	0.83	0.87	567

TABLE 4.8. Linear SVC Classifier Report: OpenSSL

Table 4.8 shows the metrics for the model. Recall is at 72% for the model. However, the precision drop shows that this model would classify more non-vulnerable items as vulnerable compared to some other tested models.

4.3.4. SGD Classifier: OpenSSL

Another model tested against the OpenSSL dataset was an SGD classifier. In previous tests, this model produced good results in terms of accuracy and precision. Figure 4.22 shows the confusion matrix; this model breaks the 50% goals for a model by 8%. The majority of the vulnerable functions were classified correctly. Around 19% of the safe functions were

misclassified as safe, which accounts for 100 functions. The SGD performed better than the decision tree when trying to detect vulnerabilities, but worse than the random forest algorithm. Figure 4.23 shows the total values for the testing datasets. The matrix shows that the model was able to predict more than 50% of the vulnerabilities, compared to other tested approaches over the Linear SVC model. The tuned Linear SVC classifier preformed better than the tuned SGD classfier. This means that this model is the second best performing model overall, when tested against the OpenSSL dataset. Predicting safe functions was not bad either, given the best model configuration.

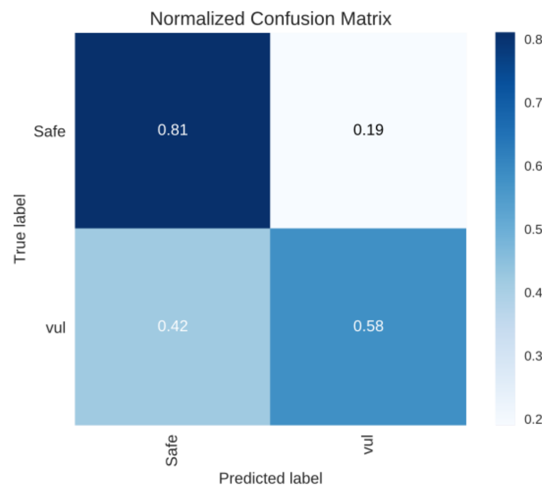

FIGURE 4.22. SGD Classifier Normalize Confusion Matrix: OpenSSL

Figure 4.24 shows the cumulative gain chart. There is a higher area under the curve using the vulnerable functions compared to the safe functions. This could have led to overall better performance.

Table 4.9 shows the metrics of evaluation for the best SGD model. In this model, the precision is still low, even though it performs worse than the best model used for classifying Vulnerabilities in OpenSSL. The recall is around 58% while the recall for the detecting safe functions is around 81%. 81% for detecting safe function is high when compared to several other models tested.

The next figure shown in this results section is the lift curve. The lift curve is shown

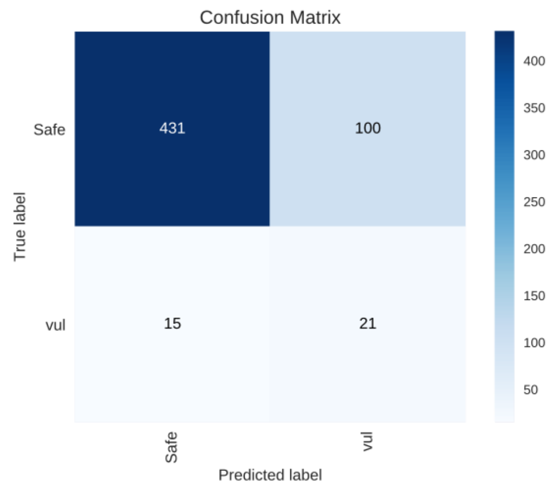

FIGURE 4.23. SGD Classifier Confusion Matrix: OpenSSL

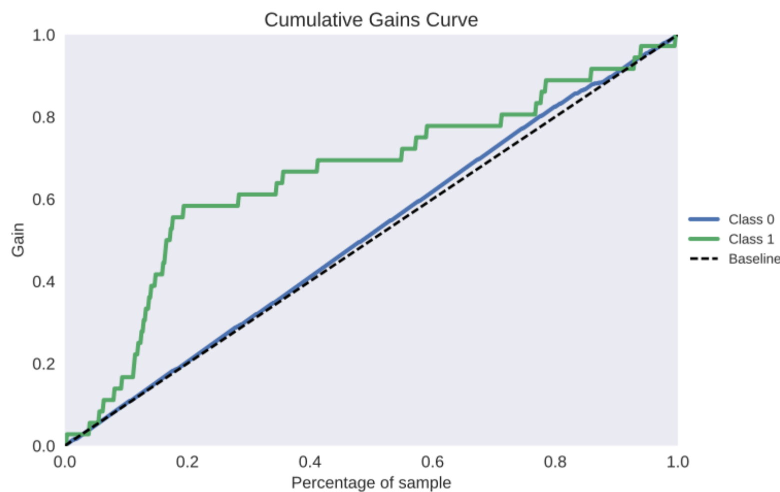

FIGURE 4.24. SGD Classifier Cumulative Gains: OpenSSL

in Figure 4.25. Ideally, a high point midway through the curve is best. However, in this curve, the lift curve has a high spike and then drops early. There is then a short rise and drop around 0.2%. The lift is fair, given the overall performance of the model.

The precision and recall curve is shown in Figure 4.26. The precision shows a sharp drop as the recall goes up. This is not the same for the safe function calculations where the

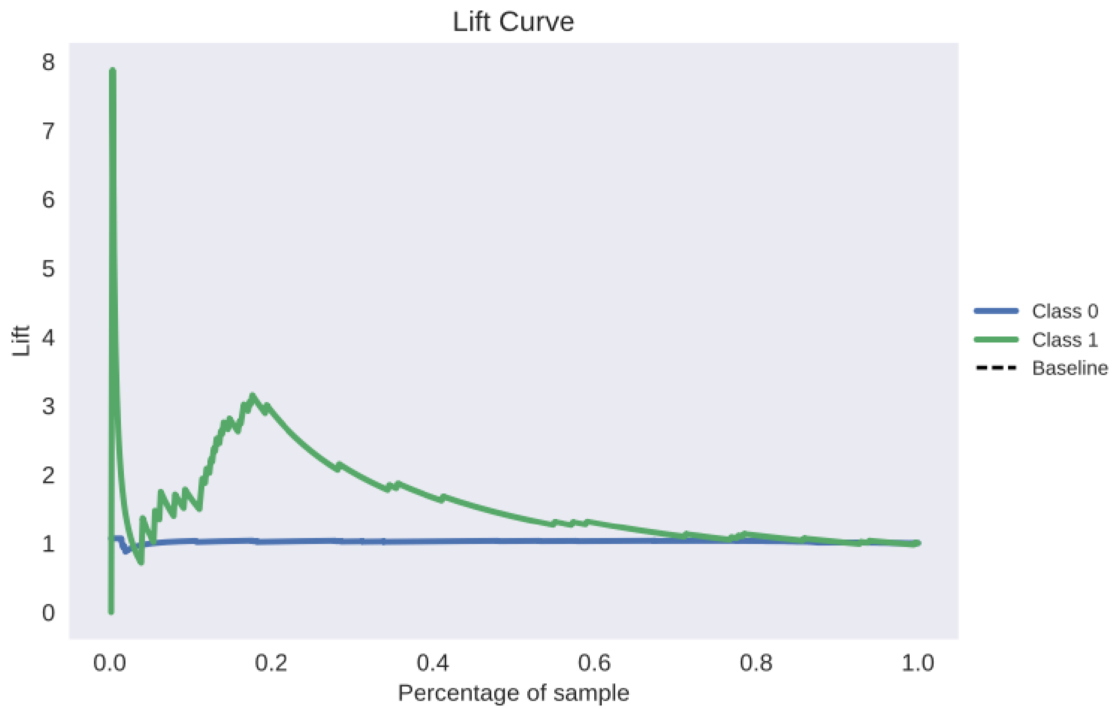

FIGURE 4.25. SGD Classifier Lift Curve: OpenSSL

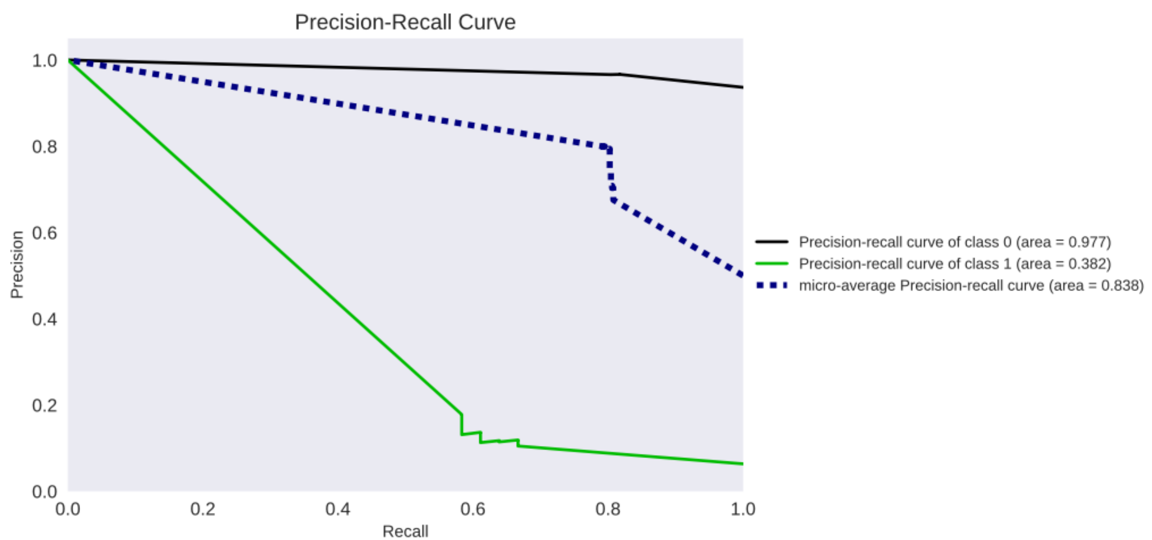

FIGURE 4.26. SGD Classifier Precision Recall Curve: OpenSSL

	precision	recall	f1-score	support
Safe	0.97	0.81	0.88	531
vul	0.17	0.58	0.27	36
avg / total	0.92	0.80	0.84	567

TABLE 4.9. SGD Classifier Report: OpenSSL

precision stays high throughout. The average is decent, although there is a sharp drop when looking at the recall and precision for the vulnerable functions.

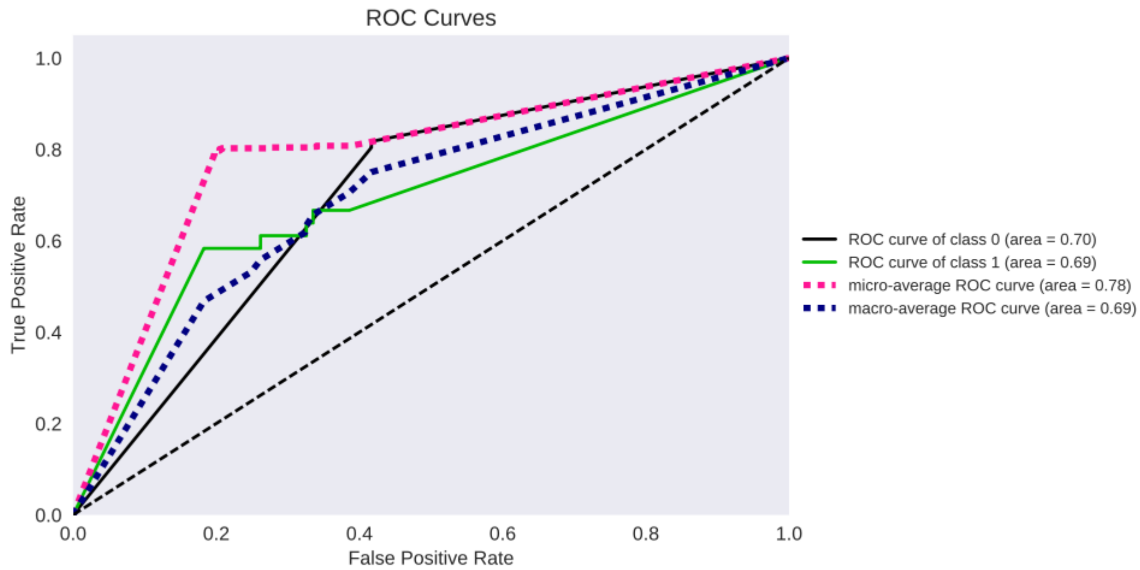

FIGURE 4.27. SGD Classifier ROC Curve: OpenSSL

Figure 4.27 shows the ROC curve. There is 1% difference between the vul and safe curves. The safe classification shows the higher percentage. The safe function curve overlaps the vul function about halfway through as shown in Figure 4.27.

4.4. OpenSSL: A Comparison of Different Bug Prediction Learning Approaches

I adapted several learning algorithms for this work. I modified algorithms by Wang, Halstead, and Nguyen in the area of bug prediction to compare these against the model using the OpenSSL dataset. I altered their feature space to the learning problem and analyzed using a grid search algorithm to find the best performing algorithm at a function/method level. I only tested linear SVC models, since I found those models to be the best performing in the prior tests. I removed features directly related to internal module level comparisons from the respective feature space. The reasoning for comparing the approaches was to determine whether this work has covered feature extraction techniques that performed better.

4.4.1. OpenSSL: Halstead Core Features

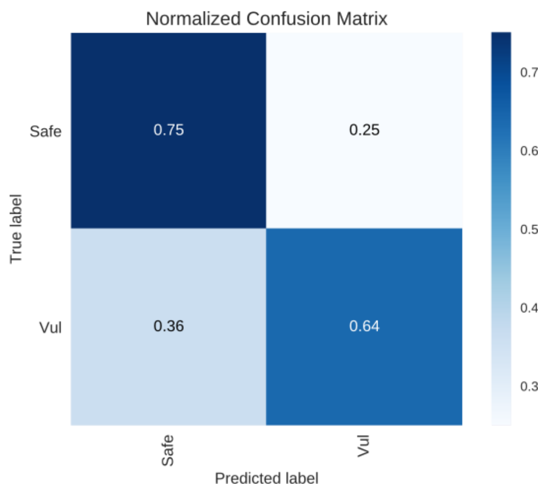

FIGURE 4.28. Halstead Core Features Linear SVC Classifier Normalize Confusion Matrix: OpenSSL

Figure 4.28 shows the normalized confusion matrix when constructing a model using a Linear SVC. The best model used the squared hinge function with a class weight that was set to balance. The model using the Halstead feature space was worse at predicting vulnerabilities, compared to this study's approach, by about 7%. Additionally, the best model was able to detect 11% more of the safe functions. Figure 4.28 also shows that using

my approach would misclassify fewer functions.

4.4.2. OpenSSL: Halstead Features

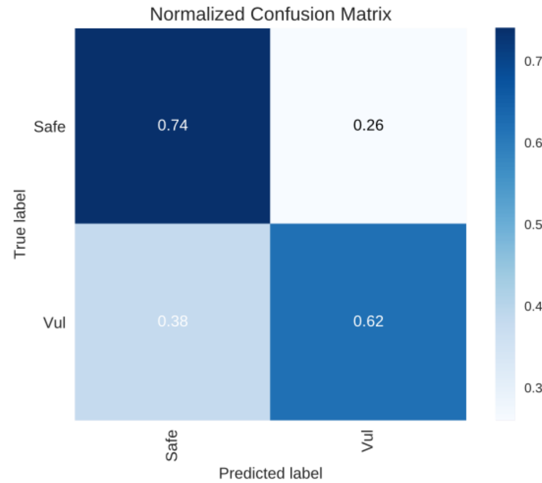

FIGURE 4.29. Halstead Features Linear SVC Classifier Normalize Confusion Matrix: OpenSSL

Halstead extended the feature space to include complexity metrics that measure program complexity. In this study, the program's complexity is focused on the change area in the LLVM IR code, which is a novel approach in comparison to those used in extant research. Figure 4.29 shows the normalized confusion matrix when constructing a model using a Linear SVC. The best model created in this work is around 10% better than those using the Halstead algorithm. Linear SVC was about 10% better at classifying safe functions. There is a drop of 1% compared to using the Halstead approach, relative to the best performing approach in the testing environment. Additionally, the best model was able to detect 11% more of the safe functions. Figure 4.29 also shows that using this approach results in misclassification of fewer feature functions. When comparing the Halstead core with the Halstead program's complexity features, the core features performed slightly better in both classifying safe and vulnerable functions. Further, using the core features improved in terms of classifying vulnerabilities. All metrics in the confusion matrix are around ±2%for

misclassifications and correct classifications.

4.4.3. OpenSSL: Nguyen

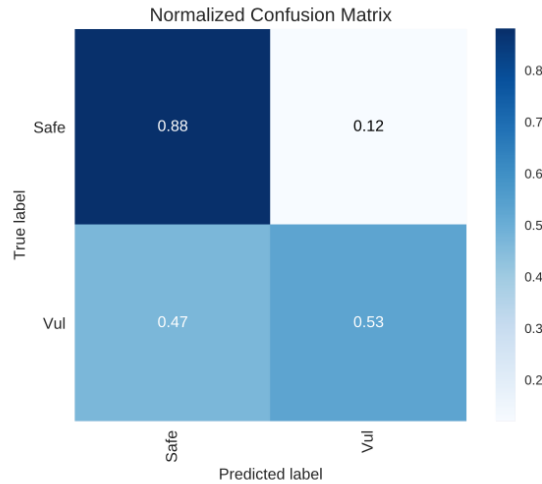

FIGURE 4.30. Nguyen Features with Linear SVC Classifier Normalize Confusion Matrix: OpenSSL

Nguyen's algorithm is based on code changes with complexity metrics. The code changes include lines, blank lines, and comment changes. Figure 4.30 shows the normalized confusion matrix Nguyen algorithm. When classifying safe functions, Nguyen's algorithm to build the feature space performs 4% better than the feature extraction technique used in this work. For classifying vulnerabilities, I was able to achieve nearly 20% better performance when classifying vulnerabilities than the Nguyen algorithm.

4.4.4. OpenSSL: Wang

The final feature extraction technique I tested was Wang's feature space, using the Linear SVC model. While Wang's features are similar to those of Nguyens model, Wang added several metrics, including blank lines, as well as introducing the number of the fan-in and fan-out calls in the function. Figure 4.31 shows the normalized confusion matrix for classifying vulnerable functions in OpenSSL using a Linear SVC. The table shows that the

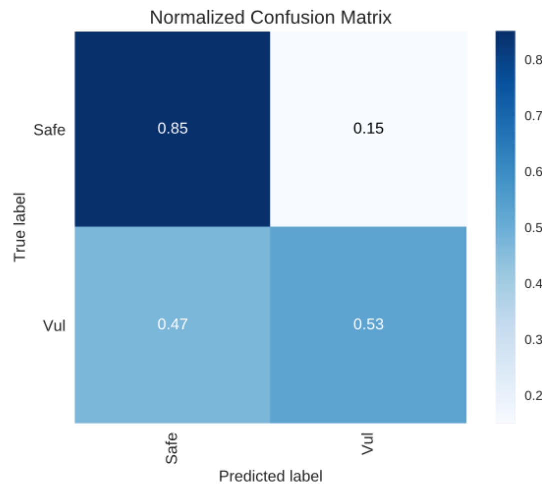

FIGURE 4.31. WangFeatures with Linear SVC Classifier Normalize Confusion Matrix: OpenSSL

model based on their feature space was able to classify 85% of the safe functions, but was only able to classify 53% of the vulnerable functions.

4.5. Summary Performance Reports

To gain a better understanding of how different groups of models were performing, I averaged difference metrics to gain an understanding of how different groups of classifiers or feature space were performing. Averaging the data across different sets shows how well given feature spaces or classifiers performed for detecting vulnerable or non-vulnerable functions. Table 4.10 shows the NIST Vul Prediction Summary. This data includes the NIST OpenSSL data and has the average precision, recall, and f1-score across all types of feature spaces. In the NIST dataset, the decision tree classifier performed the best compared to the other tree classifiers. All the tree classifiers performed better than the SGD classifier in terms of the average, precision, recall, and f1.-score. The f1-score drops by around 20% in terms of the difference between in performance the decision tree and SGD classifier. When considering the f1-score between the extra tree classifier and the random forest classifier, the extra tree classifier performed better than the random forest classifier. Extra tree classifier perform

Classifier	Avg. precision	Avg. recall	Avg. f1-score	Avg. support
Decision Tree	0.966	0.952	0.957	426.778
Extra Trees Classifier	0.948	0.931	0.938	425.111
Random Forest Classifier	0.95	0.923	0.937	425.111
SGD Classifier	0.849	0.707	0.768	426.778

TABLE 4.10. NIST Vulnerability Classification Prediction Summary

slightly better with average precision than the random forest classifier. My testing methodology allowed me to keep the average supports around 425. Support shows the number of predictions or test cases used to evaluate a given model.

Table 4.11 shows the average f1 score for the NIST data as well as all the feature spaces, with their corresponding f1-scores, for each classifier. For the decision tree classifier, using the edge op, feature space increased in performance when considering the f1-score. I identified many instances in which the f1-score increased in performance when the dataset size increased. Even the SGD classifier increased when the data was expanded from OpenSSL to the complete SARD dataset. Table 4.11 shows that the overall f1-score tends to increase in relation to the amount of data increase when building a classifier. When looking at the average across models, all feature spaces' f1-scores were above 89%, which is high. The extra trees classifier performance across features spaces is the near the same performance of the decision tree classifier.

Table 4.12 shows the summary report of the overall performance for OpenSSL with different classifiers. This table shows the average precision, recall, and f1-score for each classifier. When considering the average f1-score for determining vulnerable functions, the decision tree classifier performed the best. In terms of determining where a function was not vulnerable, all the classifiers had average f1-scores of around 97%. The high accuracy means they were effective at predicting whether a function was vulnerable or not vulnerable. All the three classifiers achieved above 60% regarding the average precision, but had low recall. However, the LinSVC was able to achieve around 72% average recall when detecting

Feature Space	DT	SGD	Extra Trees	Random Forest	avg.
Nist Openssl: edge op	0.94	0.8	0.94	0.91	0.9
Nist Openssl: edge op1	0.97	0.86	0.95	0.94	0.93
Nist Openssl: edge opn	0.99	0.89	0.95	0.96	0.95
Nist Openssl: op op	0.88	0.74	0.91	0.91	0.86
Nist Openssl: op op1	0.91	0.75	0.89	0.88	0.86
Nist Openssl: op opn	0.89	0.8	0.9	0.9	0.87
Nist Openssl: opedge op	0.96	0.83	0.92	0.94	0.91
Nist Openssl: opedge op1	0.99	0.86	0.96	0.95	0.94
Nist Openssl: opedge opn	0.99	0.9	0.97	0.95	0.95
Nist: edge op	0.95	0.75	0.93	0.93	0.89
Nist: edge op1	0.99	0.81	0.97	0.95	0.93
Nist: edge opn	0.99	0.83	0.97	0.96	0.94
Nist: op op	0.91	0.7	0.9	0.92	0.86
Nist: op op1	0.92	0.72	0.93	0.91	0.87
Nist: op opn	0.91	0.69	0.91	0.92	0.86
Nist: opedge op	0.96	0.72	0.92	0.93	0.88
Nist: opedge op1	0.99	0.81	0.96	0.96	0.93
Nist: opedge opn	0.99	0.88	0.95	0.95	0.94
avg.	0.95	0.8	0.94	0.93	

Table 4.11. SARD F1 Score Vulnerability Classification Summary

whether a function is vulnerable. SVC and SGD performed the worst in terms of detecting vulnerable functions.

Classifier	Avg. precision	Avg. recall	Avg. f1-score	Avg. support
Vul:DT	0.61	0.47	0.53	36
Vul:Extra Trees Classifier	0.62	0.28	0.38	36
Vul:LinSVC	0.23	0.72	0.35	36
Vul:Random Forest Classifier	0.78	0.19	0.31	36
Vul:SGD Classifier	0.17	0.58	0.27	36
Vul:SVC	0.44	0.31	0.36	36
Safe:DT	0.96	0.98	0.97	531
Safe:Extra Trees Classifier	0.95	0.99	0.97	531
Safe:LinSVC	0.98	0.84	0.9	531
Safe:Random Forest Classifier	0.95	1	0.97	531
Safe:SGD Classifier	0.97	0.81	0.88	531
Safe:SVC	0.95	0.97	0.96	531

TABLE 4.12. OpenSSL Vulnerability Classification Summary Report

CHAPTER 5

DISCUSSION

Using my graphical representation, feature extraction techniques, and recommended learning models, it is evident that the results generated from the NIST dataset are impressive: 72% effective in terms of finding vulnerabilities in the OpenSSL testing dataset.

5.1. Samate Dataset

The sample dataset performed very well with most learning models under investigation. Most models performed well when using simple classification labels, compared to the OpenSSL. Modifying the feature space by using different instruction classification levels,such as Op, Op1, or Opn, did not improve the performance much in the case of the NIST dataset. Further, difference in configuration did not lead to a significant increase in accuracy involving decision trees. While some models did show noticeable improvements, the performance was generally good running against the NIST dataset.

5.1.1. Vul Misclassification Problem

One major challenge in the results was trying to keep the accuracy high with different models and pre-determined vulnerabilities. It seems intuitive that models would perform better when class labels are defined based on a taxonomy. For example, if all null pointer related issues could be grouped, the performance of the models would improve, since the model would have well-defined classes. However, this assumption was proved to be false in this study; in every instance tested, using defined classes yielded low accuracy, precision, and recall.

Figures 5.1 and 5.2, and Table 5.1 show the confusion matrices and evaluation of the best model using a decision tree. The model classes are based on CWE grouping and IDs that do not fit into those CWEs. There is an issue with misclassification for the major grouping. For example, in 771 (CWE-771: Missing Reference to Active Allocated Resource), the issue occurs across the clusters: concurrency handling, memory corruption, and null pointer errors.

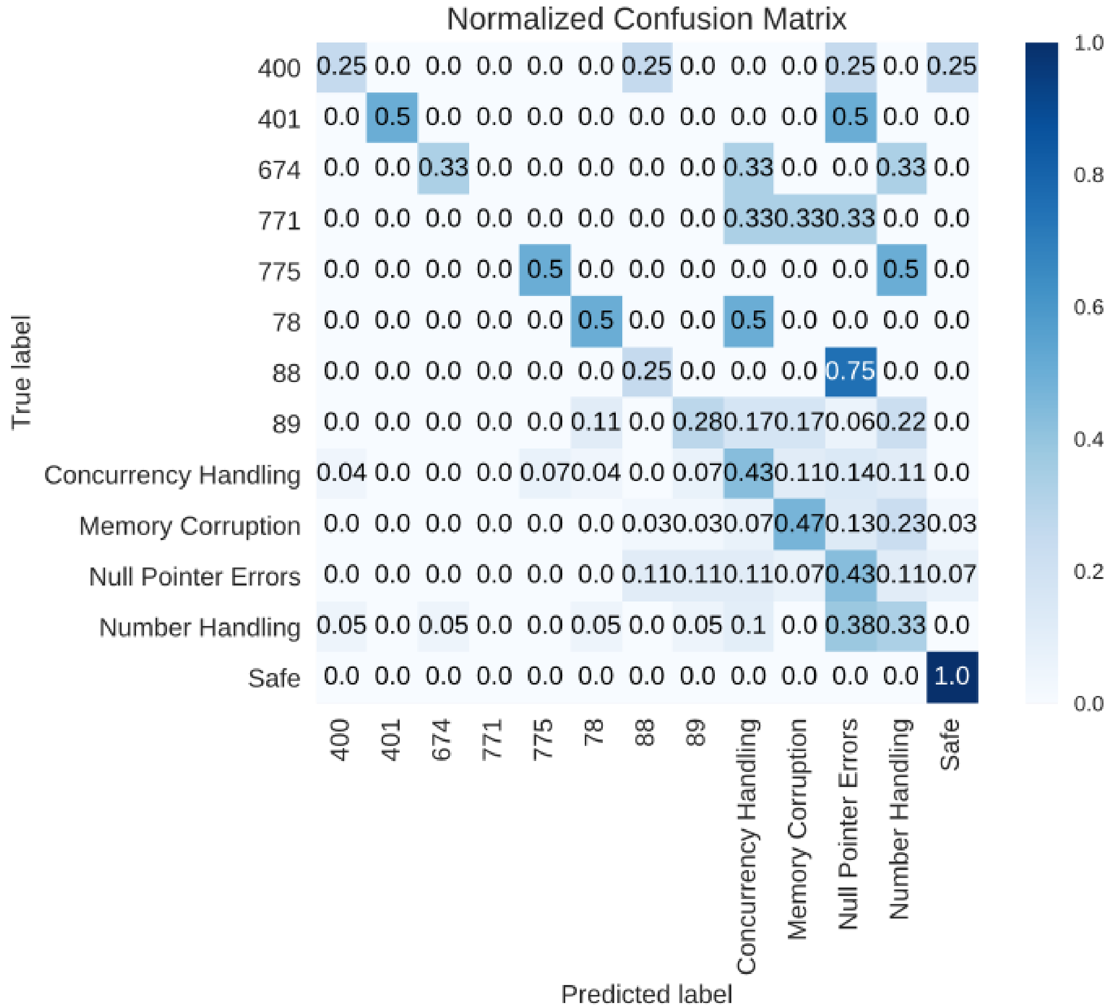

FIGURE 5.1. Normalized Confusion Matrix: NIST Dataset CWE Mapping using DT

However, there is no reason way the vulnerability classes could not be across different domains or groups. Such issue is a limitation in terms of detecting the type of vulnerability with manual analysis. Even within the groupings, there are several misclassifications.

With the issues faced when trying to build groups, it is very challenging to build an appropriate group that captures properties that do not cross domains. One suggestion would be to build bigger groups. However, as addressed in the results section, classifying based

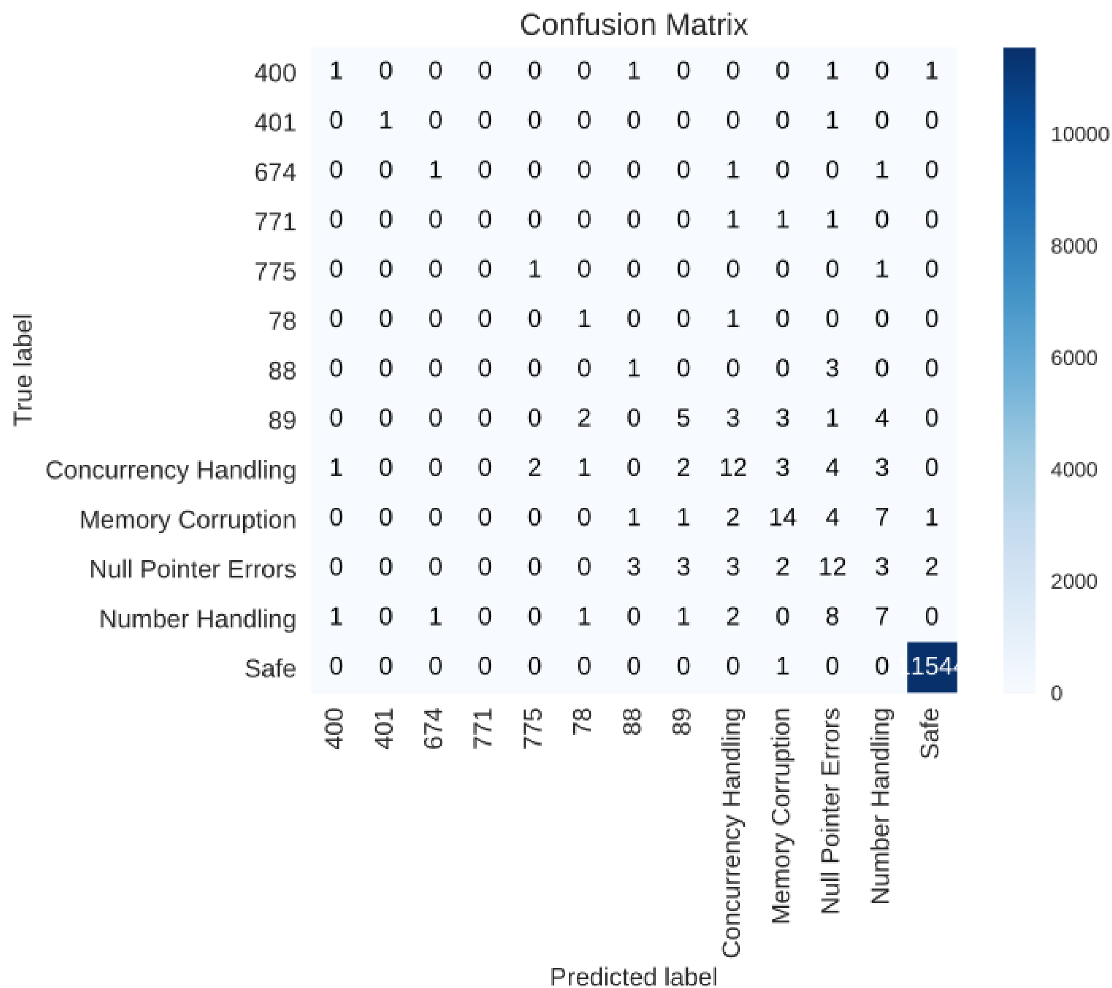

FIGURE 5.2. Confusion Matrix: NIST Dataset CWE Mapping using DT

on vulnerable properties vs. functions does not reflect how vulnerabilities work. Building subsets would just truncate to the superclass, which in the case of this research, are vulnerability functions. Based on results from many different testing and model configurations, I can conclude that the simpler the classes label, the more significant the improvement in performance.

	precision	recall	f1-score	support
400	0.33	0.25	0.29	4
401	1.00	0.50	0.67	2
674	0.50	0.33	0.40	3
771	0.00	0.00	0.00	3
775	0.33	0.50	0.40	2
78	0.20	0.50	0.29	2
88	0.17	0.25	0.20	4
89	0.42	0.28	0.33	18
Concurrency Handling	0.48	0.43	0.45	28
Memory Corruption	0.58	0.47	0.52	30
Null Pointer Errors	0.34	0.43	0.38	28
Number Handling	0.27	0.33	0.30	21
Safe	1.00	1.00	1.00	11545
avg / total	0.99	0.99	0.99	11690

TABLE 5.1. Report: NIST Dataset CWE Mapping using DT

5.2. The Pre-Post Patch Problem

During the analysis phase, I identified an issue involving determining when a function was vulnerable. This problem was deemed a pre-patch and post-patch problem. A post-patch function is the function or group of functions that changed to reflect the mitigation, while the pre-patch function is the function or group of functions involved with the vulnerability before patching the issue. I removed post-patch functions from the original dataset because the number of properties between patches is low. As previously discussed, the number of line changes is relatively low compared to the patch. Because this change is low in the overall code structure, it is difficult for a model to determine the difference between vulnerable

and non-vulnerable functions. The pre and post patch function problem can be defined as: PrePatchFunctionX and PostPatchFunctionsx. If X is the same function, the number of changes in the function is near 0. A good example of this is when a developer changes an integer value to fix an overflow issue. That small change is not captured well in the current approach. As addressed earlier in this dissertation, programs p and q are similar if they are derived from the same work. This definition also works when considering different functions. When looking for a birthmark between two functions, $s(a, b) \to [0, 1]$ is a similarity function with a value of $e < 1$. Then p and q are similar and $1 - s(a, b) < e$. Similar functions have similar properties. Thus, in most cases when dealing with pre and post patch functions, e is relatively small, unless the function is overhauled. Thus, I removed the post patch function.

5.3. Current Version of OpenSSL

The current version of OpenSSL showed the limitations of not only using synthetic data, but also some approaches that can be important for any analysis of security vulnerabilities using learning approaches. In the initial testing period, trying to classify vulnerable functions yielded low precision and recall. Modifying the classes to be binary improved the recall to around 40-50%. At that point, most models exhibited an issue with labeling many non-vulnerable functions. Ideally, to achieve a practical solution, improving the model performance is necessary. The probability of any model not classifying to the right class is higher. This problem is also referred to as over and underfitting for a given class. There are many ways to approach this problem. For example, for a random forest algorithm, the number of trees or max number of features used could be increased. The depth of the tree could also be checked. For each different sample, it would be necessary to handle the unbalanced data used to model properties. Another solution would be restricting the number of safe functions, which forces the data to balance. Although this would work, the model itself does not capture the data. While I could have used a more uniform sample but there exist up or down sampling minority classes. Down sampling removes data from the significant level to prevent that class dominant having a larger affect on the learning algorithm. Up sampling is when sampling replaces the same classes to reinforce that class. Thus, this work weight

balance the models during evaluation.

5.4. Pre-Patch Problem

One issue I encountered during analysis was a post-patch problem, also known as the post-patch paradox. This problem spans any vulnerability detection algorithms that do not rely purely on historic data or vulnerability. The paradox is that the post-patch function is no longer the sample. For all general purposes, this could be classified as a safe function. The pre-patch function is vulnerable. Thus, it can be classified as vulnerable. If learning models are built based on the vulnerability, and the features from other functions including the vulnerable function are used, the safe version of the function will be classified in the group of vulnerabilities. This can be avoided by removing overlapping properties; however, the end result would then have to handle program slices. Additionally, it may be necessary to find meta data about the program properties, because this can lead to finding more at-risk functions. The paradox is that, if a model is built with all information on the pre-patch function, it will classify the post-patch function similarly to the pre-patch function. Of course, if there are dramatic changes between the functions, this might not be an issue. However, for the general vulnerability patch, this constitutes a few lines of code. When there are a few lines of the code that change, it is difficult to distinguish between both functions with the isolated fix. Isolating the fix might create a need to acquire more information on other function calls or other callee functions. This is a challenging problem, which I chose to address by removing the pre-patch function from the dataset. I chose this approach in order to allow the model to learn from not only the patch properties, but also the properties in the function. There are several advantages to this: the model can learn from surrounding functions, compared to being isolated. Secondly, many patches add more code to handle exception handling. Therefore, using program slicing might become a challenge for any algorithm to determine whether a function is vulnerable without domain knowledge in other functions that might be vulnerable.

5.5. Patch Limitations

In this study, I focused on intra-procedural functions rather than inter-procedural functions. This means that, if a vulnerability was caused by inter-procedural or interconnected functions, the patch isolated the change to one function and the actual vulnerable information might need to be recorded for all involved functions. This is a limitation of this work; however, this study's purpose was to find vulnerable functions, as well as function locations of code at risk for a specific vulnerability type. It is also important to note that functional concepts expand to include neighboring functions without much instrumentation. This approach can go as deep as needed to capture all the properties of the function. This is powerful, because the end user can adjust the system to meet their needs with an understanding that they would need to tune the related models. Another warning in relation to the methods section and results is that most SSL-related vulnerabilities could be isolated to an individual function rather than several functions. Of course, there can be cases where several functions or a complete program need to be rewritten; however, at that level, this solution or any other methods would be challenging for any tooling. Once the problem distance becomes so large, such as a null variable but the vulnerable is found their several pointer changes, and can go several functions deep, which most tooling solutions have issues with. Another issue involving patching relates to whether the patch was vulnerable or not. Several studies state that vulnerabilities in patches are not accurate. The way to get around this issue is to focus on patches with known vulnerabilities and related CVEs or NVDs. This information is validated and confirmed by security specialists, so it is known that the vulnerability exists. One major advantage is that, often, when a patch is created, it is linked to the NVD and provides information that conveys where the patch is in the source code. Most of this work leverages this for the learning algorithms. This information can be expanded to different applications, since this information is provided to the end user through NVD and CVE. The patches provide information on the known fix for the given vulnerability. However, given that a patch is implemented in diverse ways for different applications, another issue is capturing the created information about vulnerable function properties. This is not a problem

with the approach used in this study because my focus was on the properties around the vulnerability and the vulnerability itself.

CHAPTER 6

CONCLUSION

The findings from this study show that modeling based on vulnerabilities is a difficult problem to automate without additional research and domain knowledge. Preliminary studies have revealed that either the graph with feature extraction techniques does not span well for each vulnerable domain or vulnerabilities in source code do not model well for each vulnerable class. After testing many supervised learning models and configurations, my results indicate that identifying the best way to model unknown vulnerabilities against unknown vulnerable functions is a challenging problem. However, post-patch functions need to be removed from the dataset to make any approach a practical solution. The problem found with using the pre-patch function is that they have so many properties that are similar to the vulnerability. Given that the property graphs tested are based on types, the feature extraction technique is important for determining the best learning models.

The feature extraction techniques employed in this study show the best improvement in accuracy, recall, and precision scores, compared to other models tested using features extracted from the code graph. Further, the configuration changes tested show a small effect on the overall function. While the approach used in this study involved simplifying the classification problem to only vulnerable functions and functions that had not shown any issues, the results show that many models tested had impressive performance compared to testing different classification techniques. It is harder to classify vulnerabilities among functions using information on the vulnerability than by using a binary classification method (i.e., safe vs. vulnerable). This study's approach can be applied to many security-related domains in the future to offer improved performance, compared to models that group vulnerabilities in similar clusters. The metrics collected show that a LinSVC model can be created that detects 72% of the known vulnerable functions in OpenSSL, based on historical data. In real applications, a developer or security specialist could apply the same model against a source code repository to highlight areas in the code that might be vulnerable.

This research can be expanded to any application that has vulnerable data available. Beyond the testing environment, the results of this study indicate that it is possible to automate the detection of issues in applications. This work shows a security bug finding process built into the debug process that is automated without requiring a researcher to have any domain knowledge of the applications. If an application is large enough and gets vulnerability patch reports, this vulnerability testing framework can be automated and added after NVD issues are listed. Thus, with limited user interaction, stakeholders can gain insight about the system through the analysis of their application by using learning models. The time it takes to analyze a program is relative, because after the programs are cached into graphs in the system, the process of building and evaluating models is fast. As new vulnerabilities are found, those vulnerabilities can be cached and inserted into a model and used for future analysis. The same models can be used to evaluate LTS, beta, or alpha versions of a given application when a discovery of a vulnerability is found outside the scope of normal testing. When new vulnerabilities are found, the earlier version of an application can be scanned to check for hidden vulnerable functions that have been overlooked throughout the development process. This can be done ad-hoc or during a weekly build. The entire process can be accomplished during the development process to gain a better understanding of the application and possible vulnerable areas of the code.

In tests against other feature extraction algorithms used to build models, my implementation outperformed all other algorithms for bug prediction approaches. In a few cases, my model was able to achieve more than 20% improvement compared to other bug prediction techniques. My approach outperforms tested models when predicting whether a function is vulnerable. The results of the study also indicate that bug prediction is strongly dependent on not only edge relationship in graphs but also how nodes are determined, the importance of model selection, and feature extraction techniques. Models that also used graph properties closely related to the nodes and edges performed better than algorithms that focused on complexity properties.

CHAPTER 7

FUTURE WORK

Traditional machine learning techniques were used for most of the study. Due to the scope of this dissertation, different learning techniques were not explored. Future studies should focus on the exploration of current teachings in complex neural networks. Neural networks would also work for users wanting to avoid the process of decomposition graphs using past data and general tricks to evaluate the system. The extra information could also be used to adjust the neural network. This could have overall improved results compared to not using normal networks.

One aspect of this research that was stress was the fact that the learning model is a search for at-risk functions rather than vulnerable functions. This means that what the models find is not necessarily vulnerable functions, but functions that have properties of pre-patch vulnerable functions in the source code. For this work, the focus was on the feature extraction, model construction, and model tuning. Deep explorations were not done on the causes of vulnerabilities, how the performance of the mode could be manually improved, and also if misclassifications were also a vulnerability. Future work will detail out key vulnerabilities, classify them, determine if there are vulnerability. This would require an analysis of each misclassification and vulnerability to see if they are indeed vulnerable and if the classification is wrong. Another step would be to determine if there is a way to extend to a finding that is not domain specific to that location of the code. In many research projects, the findings are domain specific and also many times case-by-case specific to the point that not many of the techniques can be expanded to other areas. Going deeper with this approach means also being able to maintain the generalizability to other vulnerable functions and applications. This will involve creating custom models and analyzing each vulnerability.

Dot representation of a program was used to allow for the analysis to run using other languages. This helps support any analysis against the graphs. However, this approach

limited the number of properties that can be collected during graph construction. The LLVM pass was created to generate the graphs. By doing the analysis inline during the pass to analyze the program, it provided more information on the program. Thus, more work can be done to interact with this current research with the learning techniques such that it could be streamlined into the C++/C application for testing. One major advantage is that one can use the build in linting and simple cast analysis that LLVM supports to improve this analysis. This can also become features for learning models such that the results of the feature extraction and graphing techniques can improve.

For most of this research, the focus was on supervised learning. The findings showed that this is the viable solution for addressing finding vulnerabilities. However, it only touches a tip of what is possible given advancements in techniques and algorithms. Learning algorithms can be broken into 3 key areas: supervised, semi-supervised, and unsupervised learning. It would be important to see if the learning algorithm can be improved by using a semi-supervised approach in order to gain a better understanding of mis-classifications. This can all be handled through the approach and provide additional details in the program study.

Another project could be examining how one could interact with actual instruction information into the models such that models can be built off of actual instruction information without losing generalizability. Another project could provide more information to enable more accurate classification of vulnerabilities, while using learning models. This is an important to create a better understanding of the application beyond a type properties graph. The biggest challenge would be learning how to model the application in such a way that the information on the instruction can be used without having too many features that would have issues with the learning model.

The final proposed future project would be adding aliasing informing. Dependency information on the given application was captured for this research, but this is vastly different from adding alias and pointer information to the model. Though information about where the instruction was related to pointer operations was captured, the relation between points to

different parts of the program was not shown. This direction would require interprocedural analysis to be adopted to handle the issues with pointers and references to other functions. However, if this is partnered with a way to convert between types and instruction, researchers and algorithms would have a better view of the application and understand better how to resolve issues.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *"Applications of Compiler Technology,"* in Compilers: principles, techniques, and tools, 2nd ed., Massachusetts: Addison–Wesley, 2007, pp. 1–35.

[2] N. Ayewah and W. Pugh, *"Null dereference analysis in practice,"* 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2010), ACM, Ontario, Canada, 2010, pp. 1–7

[3] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, *"Identifying the characteristics of vulnerable code changes: an empirical study,"* in Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), ACM, Hong Kong, China, 2014, pp. 257-268.

[4] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, *"Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations,"* IEEE Symposium on Security and Privacy (SP 2014), IEEE, California, United States, 2014, pp. 114–129.

[5] S. Cesare and Y. Xiang, *"Taxonomy of Program Features,"* in Software Similarity and Classification, Springer-Verlag: Springer, 2012, pp. 7–16.

[6] B. Chess and J. West, *"Static Analysis Internals,"* in Secure programming with static analysis, London: Addison–Wesley, 2013, pp. 72–109.

[7] L. N. Chu, *"Metric Learning for Software Defect Prediction,"* M.S. thesis, College of Information Technology, Monash Univ., Melbourne, Australia, 2015.

[8] D. Cotroneo, R. Pietrantuono, S. Russo, and K. S. Trivedi, *How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation,* Journal of Systems and Software, vol. 113, pp. 27–43, March 2016.

[9] M. D'Ambros, *"On the evolution of source code and software defects,"* Ph.D. dissertation, Università della Svizzera Italiana, Lugano, Switzerland, 2010.

[10] M. D'Ambros, M. Lanza, and R. Robbes, *"Evaluating defect prediction approaches: A*

benchmark and an extensive comparison," Empirical Software Engineering, vol. 17, no. 4–5, pp. 531–77, 2012.

[11] A. Delaitre, V. Okun and E. Fong, *"Of Massive Static Analysis Data,"* 2013 IEEE 7th International Conference on Software Security and Reliability Companion (SERE-C), IEEE, Gaithersburg, MD, 2013, pp. 163–167.

[12] W. Dietz, P. Li, J. Regehr, and V. S. Adve, *"Understanding integer overflow in C/C++,"* 2012 34th International Conference on Software Engineering (ICSE 2012), IEEE, Zurich, Switzerland, 2012, pp. 760–770.

[13] Z. Durumeric et al. *"The matter of heartbleed,"* 2014 Conference on Internet Measurement Conference (IMC 2014), ACM, Vancouver, BC, Canada, 2014, pp. 475-88.

[14] R. Eklind, *"Compositional Decompilation using LLVM IR,"* Bachelor Programme in Computer Science Thesis, Department of Information Technology, Uppsala University, Uppsala, 2015.

[15] A. Farzan, Z. Kincaid, and A. Podelski, *"Inductive data flow graphs,"* in Proceedings of the 40th annual ACM SIGPLAN–SIGACT symposium on Principles of programming languages (POPL 2013), Rome, Italy, 2013, pp. 129–142.

[16] J. S. Foster, M. W. Hicks, and W. Pugh, *"Improving software quality with static analysis,"* 2007 7th ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2007), ACM, San Diego, California, USA, 2007, pp. 83–84.

[17] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, *"Has the bug really been fixed?,"* 2010 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), IEEE, Cape Town, South Africa, 2010, pp. 55–64.

[18] T. Guelzim and M. S. Obaidat,*"Formal methods of attack modeling and detection,"* Modeling and Simulation of Computer Networks and Systems, 1th ed., Boston: Morgan Kaufmann, pp. 841–860, 2015.

[19] A. Habib, *"Finding concurrency bugs using graph-based anomaly detection in big*

code," in Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2016), Amsterdam, Netherland, 2016, pp. 55—56.

[20] M. A. Hays, *"A Fault-Based Model of Fault Localization Techniques,"* Doctor of Philosophy, Department of Engineering, Univ. of Kentucky, Lexington, KY, 2014.

[21] B. He et al., *"Vetting SSL Usage in Applications with SSLINT,"* 2015 IEEE Symposium on Security and Privacy (SP 2015), IEEE, San Jose, CA, 2015, pp. 519-534.

[22] P. Hooimeijer and W. Weimer, *"Modeling bug report quality,"* 2007 22nd IEEE/ACM international conference on Automated Software Engineering (ASE 2007), IEEE, Atlanta, Georgia, USA, 2007, pp. 34–43.

[23] D. Hovemeyer, J. Spacco, and W. Pugh, *"Evaluating and tuning a static analysis to find null pointer bugs,"* 2005 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2005), ACM, Lisbon, Portugal, 2005, pp. 13–19.

[24] M. L. Soffa and W. Le, *"Generating analyses for detecting faults in path segments,"* 2011 International Symposium on Software Testing and Analysis (ISSTA 2011), ACM, Toronto, Ontario, Canada, pp. 320–330.

[25] F. Li and V. Paxson, *"A large-scale empirical study of security patches,"* 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017), ACM, Dallas, Texas, USA, 2017, pp. 2201–2015.

[26] Z. Li, L. Tan, Z. Wang, S. Lu, Y. Zhou, and C. Zhai, *"Have things changed now? An empirical study of bug characteristics in modern open source software,"* 2006 1st workshop on Architectural and system support for improving software dependability (ASID 2006), ACM, San Jose, California, 2006, pp. 25–33.

[27] H. Liang, L. Wang, D. Wu and J. Xu, *"MLSA: A static bugs analysis tool based on LLVM IR,"* 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2016), IEEE, Shanghai, 2016, pp. 407–412.

[28] H. Liang, Q. Zhao, Y. Wang and H. Liu, *"Understanding and detecting performance*

and security bugs in IOT OSes," 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2016), IEEE, Shanghai, 2016, pp. 413–418.

[29] B. Liu, Z. Qi, B. Wang and R. Ma, *"Pinso: Precise Isolation of Concurrency Bugs via Delta Triaging,"* 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME 2014), IEEE, Victoria, BC, 2014, pp. 201–210.

[30] H. Liu, Y. Chen, and S. Lu, *"Understanding and generating high quality patches for concurrency bugs,"* 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2016), ACM, Seattle, WA, USA, 2016, pp. 715–726.

[31] B. C. Lopes and A. Rafael, *Getting Started with LLVM Core Libraries.*, 1th ed., Birmingham, UK: Packt Publishing, 2014.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou, *"Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics,"* 2008 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII), ACM, Seattle, WA, USA, 2008, pp. 329–339.

[33] Schuh, M. Dowd, and J. McDonald, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities,* 1th ed., Boston, MA: Addison–Wesley, 2006.

[34] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld and M. Jazayeri, *"Challenges in software evolution,"* 2005 8th International Workshop on Principles of Software Evolution (IWPSE'05), IEEE, Lisbon, Portugal, 2005, pp. 13–22.

[35] T. Menzies, J. Greenwald and A. Frank, *"Data Mining Static Code Attributes to Learn Defect Predictors,"* IEEE Transactions on Software Engineering, IEEE, vol. 33, no. 1, pp. 2–13, 2007.

[36] B. Meyer, *"Ending null pointer crashes,"* Communications of the ACM, ACM, vol. 60, no. 5, pp. 8–9, 2017.

[37] J. Nam, *"Survey on Software Defect Prediction,"* pp. 1–34,[Online]. Available: https://lifove.github.io/files/PQE_Survey_JC.pdf. [Accessed Oct. 31, 2018].

[38] C. N. Darin, M. Kimberly, and G. Deepa, *Modern Compiler in Java,* 1st ed., New York, NY: Cambridge University Press, 1997.

[39] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, *"Predicting vulnerable software components,"* 2007 14th ACM conference on computer and communications security (CCS 2007), ACM, Alexandria, Virginia, USA, 2007, pp. 529–540.

[40] V. H. Nguyen and L. M. S. Tran, *"Predicting vulnerable software components with dependency graphs,"* 2010 6th International Workshop on Security Measurements and Metrics (MetriSec 2010), ACM, Bolzano, Italy, 2010, pp. 1–8.

[41] C. Parnin and A. Orso, *"Are automated debugging techniques actually helping programmers?"* 2011 International Symposium on Software Testing and Analysis (ISSTA 2011), ACM, Toronto, Ontario, Canada, 2011, p. 199–209.

[42] M. Piancó, B. Fonseca and N. Antunes, *"Code Change History and Software Vulnerabilities,"* 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W 16), IEEE, Toulouse, France, 2016, pp. 6–9.

[43] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, *"Rx: Treating bugs as allergies safe method to survive software failures,"* 2005 20th ACM symposium on Operating systems principles (SOSP 2005), ACM, Brighton, United Kingdom, pp. 235–248.

[44] R. Scandariato, J. Walden, A. Hovsepyan and W. Joosen, *"Predicting Vulnerable Software Components via Text Mining,"* 2016 25th International Conference on Compiler (CC 2016), ACM, Barcelona, Spain, 2016, pp. 993–1006.

[45] Y. Sui and J. Xue, *"SVF: interprocedural static value-flow analysis in LLVM,"* 2016 25th International Conference on Compiler Construction (CC 2016), ACM, Barcelona, Spain, 2016, pp. 265–266.

[46] P-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining,* Boston, MA, Addison–Wesley, 2005.

[47] H. Wang, *"Software Defects Classification Prediction Based On Mining Software Repository,"* M.S. thesis, Department of Information Technolog, Uppsala University, Uppsala, Sweden, 2014.

[48] S. Wang, T. Liu and L. Tan, *"Automatically Learning Semantic Features for Defect Prediction,"* 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE 2016), IEEE, Austin, TX, 2016, pp. 297–308.

[49] W. Weimer, *"Patches as better bug reports,"* 2006 5th international conference on Generative programming and component engineering (GPCE 2006), ACM, Portland, Oregon, USA, pp.181–190.

[50] F. Yamaguchi, *"Pattern-Based Vulnerability Discovery,"* Ph.D. dissertation, Computer Science Dept., der Georg-August University School of Science, Gottingen, 2015

[51] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, *"Chucky: Exposing missing checks in source code for vulnerability discovery,"* 2013 ACM SIGSAC conference on Computer & communications security (CCS 2013), ACM, Berlin, Germany, 2013, pp. 499-510.

[52] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairava-Sundaram, *"How do fixes become bugs?"* 2011 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE 2011), ACM, Szeged, Hungary, 2011, pp. 26–36.

[53] S. Zaman, B. Adams, and A. E. Hassan, *"Security versus performance bugs: a case study on firefox,"*, 2011 8th Working Conference on Mining Software Repositories (MSR 2011), ACM, Honolulu, HI, USA, 2011, pp. 93–102.