

2019

## Feature Set Selection for Improved Classification of Static Analysis Alerts

Kathleen Goeschel

Follow this and additional works at: [https://nsuworks.nova.edu/gscis\\_etd](https://nsuworks.nova.edu/gscis_etd)



Part of the [Artificial Intelligence and Robotics Commons](#), and the [Databases and Information Systems Commons](#)

## Share Feedback About This Item

---

This Dissertation is brought to you by the College of Computing and Engineering at NSUWorks. It has been accepted for inclusion in CCE Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact [nsuworks@nova.edu](mailto:nsuworks@nova.edu).

Feature Set Selection for  
Improved Classification of  
Static Analysis Alerts

by

Kathleen Goeschel

A dissertation submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in  
Information Assurance

College of Computing and Engineering  
Nova Southeastern University

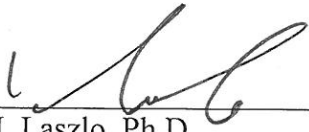
2019

We hereby certify that this dissertation, submitted by Kathleen Goeschel, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.



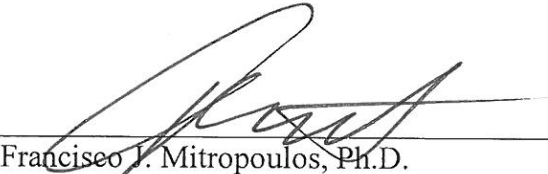
Sumitra Mukherjee, Ph.D.  
Chairperson of Dissertation Committee

Nov 20, 2019  
Date



Michael J. Laszlo, Ph.D.  
Dissertation Committee Member

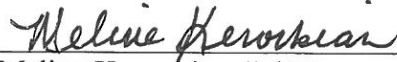
Nov 20, 2019  
Date



Francisco J. Mitropoulos, Ph.D.  
Dissertation Committee Member

Nov 20, 2019  
Date

Approved:



Meline Kevorkian, Ed.D.  
Interim Dean, College of Computing and Engineering

Nov 20, 2019  
Date

College of Computing and Engineering  
Nova Southeastern University

2019

An Abstract of a Dissertation Submitted to Nova Southeastern University  
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Feature Set Selection for Improved Classification of Static Analysis Alerts

by

Kathleen Goeschel

2019

With the extreme growth in third party cloud applications, increased exposure of applications to the internet, and the impact of successful breaches, improving the security of software being produced is imperative. Static analysis tools can alert to quality and security vulnerabilities of an application; however, they present developers and analysts with a high rate of false positives and unactionable alerts. This problem may lead to the loss of confidence in the scanning tools, possibly resulting in the tools not being used. The discontinued use of these tools may increase the likelihood of insecure software being released into production. Insecure software can be successfully attacked resulting in the compromise of one or several information security principles such as confidentiality, availability, and integrity.

Feature selection methods have the potential to improve the classification of static analysis alerts and thereby reduce the false positive rates. Thus, the goal of this research effort was to improve the classification of static analysis alerts by proposing and testing a novel method leveraging feature selection. The proposed model was developed and subsequently tested on three open source PHP applications spanning several years. The results were compared to a classification model utilizing all features to gauge the classification improvement of the feature selection model. The model presented did result in the improved classification accuracy and reduction of the false positive rate on a reduced feature set.

This work contributes a real-world static analysis dataset based upon three open source PHP applications. It also enhanced an existing data set generation framework to include additional predictive software features. However, the main contribution is a feature selection methodology that may be used to discover optimal feature sets that increase the classification accuracy of static analysis alerts.

## **Acknowledgments**

First and foremost, I would like to give my heartfelt thanks and boundless appreciation to my husband, Alan Goeschel. Your love, support, and insurmountable patience throughout this process has been a blessing. The ability for you to keep me calm in the storm exemplifies how you truly are my rock. I fully acknowledge that this process has not been easy on you – thank you. I know you are looking forward to finally getting your wife back.

I would also like to thank my advisor and committee chair Dr. Sumitra Mukherjee. Your knowledge, expertise, guidance, organization, clear direction, and responsiveness helped make this a pleasurable and enriching experience. You were always eager to guide and assist in both coursework and the dissertation process. I have learned greatly under your guidance.

A special thanks to my committee members, Dr. Laszlo and Dr. Mitropoulos, for your assistance, support, and input on this research.

I would like to thank my amazing family. Mom and Dad thank you for always supporting me, cheering me on, and guiding me throughout life's struggles. Thanks for being there to talk, hang out, and just being the amazing parents I know everyone wishes they had. I am extremely fortunate to be blessed with such amazing parents.

To my daughter Christina, thank you for being such an amazing person. I love you more than you can imagine. I am so proud of the woman, and friend, you have become. I love our talks, lunches, texts, shopping adventures, and all the fun stuff we do together. I can't wait to witness all the things you do in your life – it's just the beginning of an amazing journey.

Finally, I would like to thank my late Grammy for encouraging me to always learn, question, and solve difficult problems.

## Table of Contents

<b>Abstract</b>	iii
<b>List of Tables</b>	vii
<b>List of Figures</b>	viii

### Chapters

<b>1. Introduction</b>	<b>1</b>
Background	1
Problem Statement	2
Dissertation Goal	5
Research Questions	6
Relevance and Significance	6
Barriers and Issues	7
Assumptions, Limitations and Delimitations	8
Definition of Terms	9
List of Acronyms	12
Summary	14
<b>2. Review of the Literature</b>	<b>15</b>
<b>3. Methodology</b>	<b>31</b>
Introduction	31
The Model	31
Measures	40
Data Sets	41
Experiments	56
Resources	61
Summary	62
<b>4. Results</b>	<b>63</b>
Experiment 1	63
Experiment 2	64
Experiment 3	65
Experiment 4	65
Experiment 5	66
Experiment 6	67
Research Questions Answered	68
<b>5. Conclusions and Summary</b>	<b>70</b>
Conclusions	70
Implications and Recommendations	73
Summary	74

## **Appendices**

- A.** List and Descriptions of Features 78
- B.** Experiment Result Metrics 83
- C.** GA Performance Metrics 84
- D.** Dropped Features by Experiment 88
- E.** Top Performing Feature Subsets by Experiment 89
- F.** Sonar Scan Script 93
- G.** Data Pre-Processing 95
- H.** Control Classifier – Model A 100
- I.** Genetic Feature Selection – Model B 103

## **References 108**

## **List of Tables**

### **Tables**

1. Confusion Matrix 40
2. Definitions and Metrics 40
3. Data Set / Test Suite Requirements Matrix 45
4. Raw Data Set Alert Statistics 56
5. Pre-Processed Data Set Alert Statistics 58
6. Final Data Set Alert Statistics 58
7. Experiment, Project, and Data Set Used 60
8. Experiment 1 Results 64
9. Experiment 2 Results 64
10. Experiment 3 Results 65
11. Experiment 4 Results 66
12. Experiment 5 Results 67
13. Experiment 6A Results 68
14. Experiment 6B Results 68



## **List of Figures**

### **Figures**

1. GA Crossover 33
2. Possible Hyperplanes 37
3. Linear SVM Hyperplane 37
4. Framework for Static Analysis Alert Generation and Labeling 55
5. Model Training, Testing, and Analysis 61

## Chapter 1

### Introduction

#### Background

Static analysis (SA) tools analyze source code to find flaws and defects without the need to execute the binaries. Static application security testing (SAST) is an extension of traditional static analysis that focuses on discovering security vulnerabilities either by analyzing source code or the binaries. Historical research in the literature began with static code analysis (SCA) to locate bugs (Graves, Karr, Marron, & Siy, 2000; Johnson, 1978; Munson & Khoshgoftaar, 1992; Ostrand, Weyuker, & Bell, 2004). The domain has evolved to include security vulnerability testing (Chen & Wagner, 2002; Chess & McGraw, 2004; Evans & Larochelle, 2002); however, some SA tools find only traditional bugs or flaws, some find only security vulnerabilities, and some include both. Since some traditional bugs and flaws could be classified as security vulnerabilities there can be some overlap in what these tools report. Therefore, for the duration of this paper, reference to both SCA and SAST will be collectively referred to as static analysis (SA); the static scanning of code or binaries in order to locate bugs, defects, flaws or security vulnerabilities.

For both types, there exist commercial and open-source tools. There are reasons why multiple tools would be used in conjunction. Different tools may have increased accuracy at detecting a specific type of defect (Nunes et al., 2017). Some tools are language specific while others may be capable of processing several languages. Employing

multiple static analysis tools can increase defect detection (Wang, Meng, Zhou, Li, & Mei, 2008; Wedyan, Alrmuny, & Bieman, 2009). Additionally, confidence in the alert may be increased if multiple tools alert on the same item (Muske & Serebrenik, 2016).

SA tools may functionally operate in different manners. Typically, SA tools make an abstraction of the program by mapping variables, functions, methods, states, files, inputs, outputs, etc. How they accomplish this feat of abstraction can include one or several methodologies such as lexical analysis, model checking, control flow analysis, data flow analysis, symbolic analysis, information flow analysis, or taint analysis (Zhioua, Short, & Roudier, 2014a, 2014b). Most tools create an abstract syntax tree (AST) to represent the program and to map program flow.

Who uses SA tools and at what point in the software development process they are leveraged varies between projects, organizations, and roles. SA tools are utilized by developers, security analysts, or other persons involved in the software development or operations process. The scans may be initiated via the tool running on a server, local machine, command line, automated script, an integrated development environment (IDE), or a continuous integration (CI) pipeline. Additionally, a user's interaction with SA tools often depends on a person's role in the development process.

## **Problem Statement**

Software applications are used in innocuous applications such as games and entertainment as well as in critical or sensitive applications such as banking, electrical grids, and medical devices. Notwithstanding the criticality of an applications use, the successful attack of even an innocuous application, such as a game, could grant attackers access to sensitive data and networks.

In 2017, 48% of data security breaches involved hacking and 76% were financially motivated (Verizon, 2018). Over 60 million dollars of loss was reported due to corporate data breaches (FBI, 2017). In addition, 81% of breaches resulted in a loss of confidentiality (Verizon, 2018). The loss of confidentiality to personal identifiable information has also been evidenced by the Equifax data breach which exposed 145 million Americans personally sensitive information (US Senator Elizabeth Warren, 2018) and in the Facebook data breach exposing 87 million Americans personal information (Badshah, 2018). With the extreme growth in third party cloud applications (Cisco, 2017), increased exposure of applications to the internet, and the impact of successful breaches, improving the security of software being produced is imperative.

The detection and remediation of security vulnerabilities early in the development lifecycle is less costly to correct than post development phases (Ayewah & Pugh, 2010; Bishop, Gashi, Littlewood, & Wright, 2007; Chess & McGraw, 2004; Ogasawara, Aizawa, & Yamada, 1998). SA tools can alert to quality and security vulnerabilities of an application; however, they present developers and analysts with a high rate of false positives and unactionable alerts (Goseva-Popstojanova & Perhinschi, 2015; Johnson, Song, Murphy-Hill, & Bowdidge, 2013). Additionally, ambiguity in prioritization schemas make the task of determining which alerts to address first confusing for developers and analysts (Kim & Ernst, 2007b). Valuable time and effort is wasted when analyzing irrelevant alerts (Beller, Bholanath, McIntosh, & Zaidman, 2016). This problem may lead to the loss of confidence in the scanning tools, possibly resulting in the tools not being used (Johnson et al., 2013; Reynolds et al., 2017). The discontinued use of these tools may increase the likelihood of insecure software being released into

production. Insecure software can be successfully attacked resulting in the compromise of one or several information security principles such as confidentiality, availability, and integrity.

Software defects and faults are commonly referred to as bugs. The introduction of bugs into an application can result in the failure for the code to compile, unintended behavior, or the instability or un-usability of the application. Thus, early efforts in the SA domain were focused on analyzing source code to find bugs (Hovemeyer & Pugh, 2004; Ostrand et al., 2004). Application attacks attempt to cause an application to behave in an unintended manner by specifically targeting the application layer. SA tools were expanded to detect security vulnerabilities in source code (Chess & McGraw, 2004), referred to as static application security testing. However, SAST presents challenges including high false positive rates (Goseva-Popstojanova & Perhinschi, 2015; Johnson et al., 2013), poor prioritization schemas (Carrozza, Cinque, Giordano, Pietrantuono, & Russo, 2015; Heckman & Williams, 2013), unactionable alerts (Hanam, Tan, Holmes, & Lam, 2014; Heckman, 2007), and incomplete or missing data sets for testing (Delaitre, Stivalet, Fong, & Okun, 2015).

Citing these problems, several research efforts have been made to prune, parse, mine, and prioritize the results from these tools. Pruning efforts result in the overall reduction of alerts (Chimdyalwar & Kumar, 2011; Hanam et al., 2014; Yüksel & Sözer, 2013). Parsing of the alerts and associated features have been used for clustering (Fry & Weimer, 2013; Podelski, Schäfer, & Wies, 2016). Data mining of the results have been used to find statistically significant features for predicting false positives, actionable, and unactionable alerts (Medeiros, Neves, & Correia, 2016; Ruthruff, Penix, Morgenthaler,

Elbaum, & Rothermel, 2008). Ranking schemes for the prioritization of alerts have shown success (Carrozza et al., 2015; Heckman, 2007; Kremenek & Engler, 2003; Zhang, Jin, Xing, Zhang, & Gong, 2013).

Despite these efforts, the problem of high false positive rates and irrelevant alerts persists and additional research is needed. Machine learning has been explored in the literature for this domain (Bleier, 2017; Koc, Saadatpanah, Foster, & Porter, 2017; Pang, Xue, & Wang, 2017); however, the work is limited. In-depth investigation into feature selection for improved alert classification will greatly add to the literature in the static analysis domain.

### **Dissertation Goal**

Machine learning has successfully been used to classify items from other domains such as in intrusion detection systems (Buczak & Guven, 2015), financial systems (Heaton, Polson, & Witte, 2017; Sun & Vasarhelyi, 2018), medical diagnosis (Hussain, Aziz, Saeed, Rathore, & Rafique, 2018; Kourou, Exarchos, Exarchos, Karamouzis, & Fotiadis, 2015), as well as in the static analysis domain (Bleier, 2017; Koc et al., 2017; Pang et al., 2017).

Feature selection is the process of determining which features are relevant or will improve classification accuracy (Xue, Zhang, Browne, & Yao, 2016). Feature selection prior to classification has shown to improve classification (Ambusaidi, He, Nanda, & Tan, 2016; Xue et al., 2016).

In the SA domain, there lacks consensus as to what features are relevant (Bell, Ostrand, & Weyuker, 2006; Heckman, 2007; Ruthruff et al., 2008; Shivaji, Whitehead, Akella, & Kim, 2013). Additionally, relevant features may differ between projects

(Heckman & Williams, 2009). Knowledge of which features to focus on could allow more effort to be placed on gathering those relevant features.

Although feature selection methods have previously been applied to the SA domain with success (Bell et al., 2006; Heckman & Williams, 2009; Ruthruff et al., 2008; Shivaji et al., 2013), the work is limited, sometimes conflicting, and recent advances in feature selection methodologies could improve results.

Therefore, feature selection methods have the potential to improve the classification of static analysis alerts and thereby reduce the false positive rates. Thus, the goal of this research effort was to improve the classification of static analysis alerts by proposing and testing a novel method leveraging feature selection.

The proposed feature selection model's performance was tested against a similar model that utilized all features. Accuracy, precision, recall, and the false positive rate were used to compare the two models' performance.

## **Research Questions**

The research questions that guided this effort and were answered as a result include:

1. Does the proposed model improve the classification of alerts?
2. Do selected feature subsets from the proposed model vary between projects?
3. Are some features never selected?
4. Similarly, are some features always selected?

## **Relevance and Significance**

In recent years the importance of application security has increased due to high profile data breaches (Badshah, 2018; Verizon, 2018; US Senator Elizabeth Warren, 2018), and

increased cloud computing (Cisco, 2017). Successful attacks can impact a company's reputation and can result in financial loss (FBI, 2017). SA tools can alert to quality and security vulnerabilities of an application; however, they present developers and analysts with a high rate of false positives and unactionable alerts (Goseva-Popstojanova & Perhinschi, 2015; Johnson et al., 2013). There has been much research effort dedicated to the improvement of both the SA tools performance and accuracy as well as researching methods to process the data output of these tools. This research effort focused on processing the output of these tools and is relevant to today's application security challenges.

As a result of this research effort feature sets that are relevant for accurate classification of static analysis alerts were discovered.

## **Barriers and Issues**

There were some challenges to this endeavor. Complete data sets are lacking in the domain (Heckman & Williams, 2008; Herter, Daniel, Mallon, Wilhelm, & GmbH, 2017; Shiraishi, Mohan, & Marimuthu, 2015). However, a framework cited in the domain literature was followed to generate data sets (Heckman & Williams, 2008, 2009). This process was followed; however, it added additional complexity and time to the overall research task. Auxiliary information from systems were downloaded and queried such as vulnerability disclosures and release notes. Custom scripts were written, tested, and utilized to perform code scanning, gather and process the static analysis alerts, track alerts through versions, match alerts with features, and automatically label alerts.

Static analysis tools lack consistency in their abilities and outputs. There exist a multitude of programming languages varying in syntax and semantics. Consequently, SA



tools must be designed to parse and process a particular language in order to perform a scan. Thus, some tools have the capability to scan multiple languages while other tools are limited to a few, if not one. None of the tools scan all languages. Additionally, SA tools output may widely vary from simple file, line of code, and alert type to robust information such as confidence, shared sinks, history, and data flow path. Consequently, the feature sets utilized in static analysis are often heterogenous. All of the scanners selected for this research were capable of scanning the selected languages and offered similar outputs; nevertheless, the resulting output from each tool was reviewed and features that were not provided in every tool were removed from the data set.

### **Assumptions, Limitations and Delimitations**

The quantity of alerts to manually label was not feasible; therefore, a framework for automatic labeling of alerts was used (Heckman & Williams, 2008, 2009). A few assumptions were made during this process. Alerts that disappeared from one version to another, not due to file deletion or an easily detected file rename, was assumed to be fixed (a true positive). It was possible that the alerts disappeared for different reasons between versions; however, those reasons and their possible detections were beyond the scope of this work. Alerts that disappeared due to a deleted file were disregarded as there was no way to determine if the deletion was intended to correct the error (i.e., a complete re-write of a component) or for other reasons. These assumptions are consistent with previous works (Bleier, 2017; Heckman & Williams, 2008, 2009; Yan et al., 2017).

Additionally, a limitation of this study is that only one main programming language was used in the test applications. It is possible that the feature selection model proposed may select different feature subsets for different programming languages. However, what

is presented within is a model that may be used to select relevant feature sets per project, programming language, or time frame.

### **Definition of Terms**

**Accuracy:** the proportion of correctly classified instances.

**Actionable Alert:** an alert in which action has been taken to resolve the alert.

**Abstract Syntax Tree:** a representation of source code as a tree structure.

**Alert Characteristics:** the features of an alert.

**Alert Lifetime:** the time from which an alert appears to when it disappears.

**Artificial Data Set:** a data set created with artificial data.

**Benchmark:** a test suite developed by OWASP to evaluate static analysis tools.

**Chromosome:** in genetic algorithms, a possible solution set.

**Churn:** a source code metric for number of lines added, modified, and deleted.

**Classification:** a machine learning process in which inputs are predicted to belong to a particular class.

**Comma-Separated Values:** a type of file that organizes data in rows with columns separated by delimiters.

**Common Weakness Enumerations:** a list of common security weaknesses.

**Concurrent Versioning System:** a program that creates a common repository for source code that tracks versioning and changes. Allows multiple developers to share and modify the source code without overwriting each other's changes.

**Confusion Matrix:** a table used to present the performance of a classification model.

**Continuous Integration:** a software development practice by which upon software code being committed, a build is performed to assure the newly added code does not break in the desired environment or fail to build completely.

**Cross-Site Scripting:** an application security vulnerability that allows attackers to inject client-side scripts.

**Crossover:** in genetic algorithms, the process by which two parents generate offspring.

**Cyclomatic Complexity:** a source code metric for the number of paths through a function.

**Data Set:** a set of data that includes inputs (features) and expected outputs (labels).

**Deep Neural Network:** a neural network comprised of three or more layers.

**Drupal:** an open source web-based content management system.

**Dynamic Analysis:** the process of testing applications for defects by executing the program in real-time.

**Engineered Feature:** a new feature derived or calculated from an existing feature or set of features.

**F-Measure:** the harmonic mean of precision and recall.

**Fan-In:** a source code metric for number of functions calling a function.

**Fan-Out:** a source code metric for number of functions called by a function.

**Feature Selection:** the process by which a subset of relevant predictive features is selected from a full feature set.

**Feature:** an attribute shared by all the independent instances upon which learning may be performed.

**FindBugs:** an open source static analysis program that finds bugs in Java code.

**FindSecBugs:** an open source static analysis program that finds security vulnerabilities in Java code.

**Fitness Function:** in genetic algorithms, a function used to evaluate the performance of a solution on the input.

**Genetic Algorithm:** an algorithm that uses natural selection and evolutionary methods to find an optimal solution for inputs by optimizing for an objective function.

**Integrated Development Environment:** a software tool used for software development.

**JAVA:** a compiled programming language that is operating system agnostic.

**JULIET:** a test suite of test cases developed by NIST to evaluate static analysis tools.

**Long Short-Term Memory:** a machine learning network comprised of recurrent neural networks.

**Machine Learning:** the computational process of building models based upon learning patterns in input data.

**MITRE:** a non-for-profit research company.

**Moodle:** an open source web-based learning management system.

**Mutation:** in genetic algorithms, a random change in a chromosome to promote diversity, similar to biological mutation.

**National Vulnerability Database:** The United States Government's repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (NIST, 2017b).

**Neural Network:** a machine learning method based on the concepts of the human brain.

**PHP:** a programming language for web-based applications.

**PhpMyAdmin:** an open source web-based database administration tool for MySQL.

**Precision:** the proportion of true positives classified correctly.

**Preprocessing:** the preparation and transformation of data for machine learning.

**Recall:** the proportion of true positives correctly classified as positives.

**SonarQube:** an open source static analysis program that finds security vulnerabilities and code quality issues for several languages.

**Static Analysis:** the process for testing applications source code or binaries for bugs or flaws without executing the application.

**Static Application Security Testing:** the process for testing applications source code or binaries for security vulnerabilities without executing the application.

**Structured Query Language Injection:** a security vulnerability that allows attackers to inject queries on a data source.

**Support Vector Machine:** an algorithm for machine learning classification.

**Test Suite:** a collection of source code with labeled good and bad test cases that could be used to create labeled data sets.

**Testing Data Set:** a subset of a data set used to evaluate a machine learning model.

**Training Data Set:** a subset of a data set used to train a machine learning model.

**Unactionable Alert:** an alert that persists between versions.

### **List of Acronyms**

**AC:** Alert Characteristics

**ARM:** Adaptive Ranking Model

**AST:** Abstract Syntax Tree

**BLOC:** Blank Lines of Code

**CAS:** Center for Assured Software

**CI:** Continuous Integration

**CIT:** Chrome Issues Tracker

**CLOC:** Commented Lines of Code

**CSV:** Comma-Separated Values

**CVS:** Concurrent Versioning System

**CWE:** Common Weakness Enumerations

**DNN:** Deep Neural Network

**FN:** False Negative

**FP:** False Positive

**GA:** Genetic Algorithm

**HTML:** Hypertext Markup Language

**IDE:** Integrated Development Environment

**JSON:** JavaScript Object Notation

**LOC:** Line of Code

**LSTM:** Long Short-Term Memory

**MFSA:** Mozilla Foundation Security Advisor

**NL:** Number Lines

**NN:** Neural Network

**NSA:** National Security Agency

**NVD:** National Vulnerability Database

**OWASP:** Open Web Application Security Project

**OX1:** Order Base Crossover

**PHP:** Hypertext Preprocessor

**PMD:** Programming Mistake Detector

**PMX:** Partially Mapped Crossover

**POS:** Position Based (crossover)

**SA:** Static Analysis

**SARD:** Software Assurance Reference Dataset

**SAST:** Static Application Security Testing

**SCAP:** Security Content Automation Protocol

**SCX:** Sequential Constructive Crossover

**SQLi:** Structured Query Language Injection

**SVM:** Support Vector Machine

**TN:** True Negative

**TP:** True Positive

**WAP:** Web Application Protection

**XML:** Extensible Markup Language

**XSS:** Cross-Site Scripting

## **Summary**

This Chapter has outlined a brief history of the static analysis domain and its current problems, discussed motivating factors for continued research, and posited the goal of this proposed research effort. Research questions that guided this research were presented. Barriers, limitations, and assumptions were identified. The rest of this paper is organized as follows: Chapter 2 presents a review of literature; Chapter 3 outlines the research methodology that was followed.

## **Chapter 2**

### **Review of the Literature**

Research efforts into the processing of alerts generated from static analysis tools began to flourish in the early 2000's. This included ranking methodologies using statistical techniques, historical information, clustering, and feature selection for prediction models.

An alert ranking program, Z-Ranking, was proposed in (Kremenek & Engler, 2003). This program ranked error messages in order of probability based upon frequency counts of successful and failed checks. After counting the number of successful verses unsuccessful checks, the program used statistical techniques to compute and sort error messages based upon those values. They tested their solution on two systems, Linux 2.5.8 and a commercial code base. They ranked the reports for comparisons of: Z-Ranking, the tools default, random ranking, and optimal ranking. Their method outperformed the default and random for both programs in all three scenarios. It also performed better than randomized ranking 98.5% of the time. However, it never outperformed optimal thus leaving room for improvement.

Improving the ranking of alerts by using correlation between reports was pursued in (Kremenek, Ashcraft, Yang, & Engler, 2004). Report errors were grouped into sets of correlated populations based on code locality using function, file, and directory. Initial ordering for reporting the errors was performed by assigning a probability that the item was a bug. After the inspection of an item was completed and ranked, the information



gained from feedback was used to update the probabilities using a Bayesian Network. The error report was then re-prioritized to display the next error. They tested their method using error reports from Linux 2.4.1 and a large commercial system. They manually classified the error reports and were able to cluster the errors into 4 regions. The authors saw a factor of 2-8 improvement over randomized ranking showing that re-prioritization of error reports using information gain is beneficial.

A foundational work in the domain that attempted to accurately predict files that contain the largest amount of faults was presented in (Ostrand et al., 2004). The authors goal was to provide testers with a practical and reasonably accurate assessment of which files contained the largest faults. In other words, where to find the bugs. They created a negative binomial regression model using information from previous releases as predicting factors of fault probabilities. Files were ordered by descending number of predicted faults. Factors they used included log(LOC), file age, new file, changed file, number and magnitude of changes made to the file, square root of the number of detected faults in prior releases, number of faults in early development stages, programming language, and release number.

They tested their model on a large telecommunications inventory system that had 17 releases and another system that had 9 releases. Their results found that the top 20% of files from the model contained 83% of the faults. Statistically significant factors may have skewed this model to its simplest form such as lines of code (LOC); however, this work was extremely beneficial to the domain, well thought out, and executed well.

FindBugs, a widely used SA tool, was presented in (Hovemeyer & Pugh, 2004). The authors described in detail how their program used 50 bug patterns in several rough

categories and tried to locate them in source code. They tested their program on six highly used programs, including open-source and very large programs. They found several bugs and reported the bugs to the vendors. The vendors fixed the bugs in future releases; thus, clearly illustrating that their program did in fact find legitimate bugs. They compared their program with PMD, another SA tool, and found that the two were complementary and not intended to replace each other. This tool is widely used today, even as a component in commercial SA tools.

(Bell et al., 2006) extended their previous work by testing their model on a younger system using four different feature-based models. They used a variable selection process by computing statistical significance of different variables to include in their models. They created four models of different variable sets to test on the new system.

1. Basic Model: included features log(LOC); log of proportion of the month the file was in the system; the age of the file in months; indicators if the file was new, one, two, three or four months old; the square root of the number of changes made in the prior month; the square root of the total number of changes made during the last five months; indicators for js or sh; indicators of language for which the average file size was very small or small; and dummy variables for all but one month.
2. Enhanced Model: included all the features of the basic model plus dummy variables for conf, html, java, jsp, xml, and xsl; interactions of log(LOC) with each of conf, html, sh, xsl; and the very small grouping.
3. Simplified Model: used log(LOC); new to the month or not; changed or not; months in the system as files age; and log of exposure variable which was a variable for duration of the month it was in the system.
4. LOC Model: used a simple count of the number of lines of code in a file.

Their test results show that the LOC Model covered 55% of the faults, Simplified Model around 65-67%, the Basic Model 71-75%, and the Enhanced Model of 71-75%. It showed that the Basic or Enhanced models were in line with their previous research results.

Another alert ranking system was outlined in (Williams & Hollingsworth, 2005) that leveraged source code change history. They mined a concurrent versioning system (CVS) repository for source code changes. They used two factors in ranking the alerts: whether the function was previously part of a bug fix and the percentage of times a function return value was checked prior to its use. They tested their ranking system on Apache Web Server and Wine and compared their results with a naive technique: solely consisting of an indication if the return value was checked more than half the time. They were able to demonstrate that ranking criteria can be improved by mining software repository historical information.

(Kim & Ernst, 2007a) prioritized warning categories by analyzing software change history. Using the lifetime of a bug, they prioritized shorter lifetimes as higher priority and longer lifetimes as a lower priority. They analyzed two programs: Columba and jEdit. The authors ran scans on each compilable version of the code using three SA tools. The authors calculated the time between when a bug appeared to when the bug disappeared. The authors found that prioritized ranking varied between both tools and projects, and that the lifetime for each category of bug differed. The authors assumed more serious bugs were fixed first which may not always hold true. For unfixed bugs they set a default number of resolution days which could skew results since they also issued

the same priority level for all bugs of the same category. Another assumption was that all bugs reported were true positives.

A work to prioritize alerts by utilizing source code change logs from versioning history was presented in (Kim & Ernst, 2007b), an extension of their previous work. In this work, they evaluated their weighted prioritization method on three programs, Columba, Lucene, and Scarab, using three code scanner tools. They identified potentially buggy lines of code by mining change log messages for bug related keywords. They marked the changed lines from the previous versions as buggy or non-buggy. They then ran code scanners and compared the alert reports to the list generated using the change logs. Grouping was performed by category. For each warning in a category, they increased the weights by different factors if the warning was removed in a fix change verses a non-fix change. The final weight was the weight divided by the number of warnings in a category. The list was re-prioritized using the new weights. Their prioritization method improved the warning precision overall by 17%, 25%, and 67% respectively.

There are limitations to this work. For instance, all warnings in the same category were given the same weight. Additionally, they removed a warning if the file was deleted. Perhaps they could have searched for a hash of the file to verify if the file moved or changed names. Quite noteworthy, the Weighted majority voting and Winnow online machine learning algorithms cited as justification for their algorithm uses not only promotion but demotion; however, their algorithm lacks a demotion aspect. They encountered similar challenges to other works citing that bug fix data was incomplete in

the logs and matching warnings between versions was difficult due to line of code changes and the deletion of files.

Adaptively ranking static analysis alerts by using historical data from developer feedback was proposed in (Heckman, 2007). The adaptive ranking model (ARM) gathered data from three sources: the alerts generated from static analysis, developer feedback, and historical ranking factors. The author gathered the alerts generated from static analysis and then ranked the listings using their algorithm.

The author presented four equations which were utilized in the alert ranking process. The proportion of closed and suppressed alerts from the developer to all suppressed and closed alerts was used to arrive at an adjustment factor which modified ranking factors. Alert type accuracy was computed using the weighted average of historical data from observed true positive rates and the actions of suppression and closing of alerts by the developer. Code locality used the historical data based on alerts that the developer suppressed and closed in the same area of code by method, function, and folder. Finally, alert type accuracy and code locality were used for ranking the alerts.

The author tested their ranking system on iTrust, a health care Java application written as a school project at North Carolina State University, and compared their ranking system with an optimal ranking, random ranking, and a tool default ranking. The ARM performed very close to the optimal ordering of alerts and discovered 81% of the true positive alerts in the first 20% of inspections. The random ranking found only 22% in the first 20% of results. The ARM's performance then did degrade; however, it still outperformed random and eclipse.

Although this work was promising it presented with limitations. First, it was used on an unused software product. iTrust is not a real-world application so is not indicative of the types of issues found. Additionally, this was not previously used for testing in the domain. For some types of alerts, the initial weights used were taken from other published works. This model ranks alerts based upon what the developers are closing or ranking. This can create lists specific to what an organization is deeming to be most valuable to them. However, if developers choose to tackle one type of alert at a time, those alerts may get erroneously ranked higher.

From the work performed by researching the historical data it was soon discovered that some bugs persisted over time. These were soon classified as unactionable: true alerts yet not acted on by developers. Reasons for not correcting the bugs are not proven and lacks exploration in the literature.

Machine learning was used to build false positive mitigation models to classify static analysis alerts as actionable or unactionable in (Heckman & Williams, 2009). They wanted to find from the static analysis alerts a set of alert characteristics (AC) that are predictive of actionable alerts and which models are best at classifying them.

For possible ACs, the author used features from: the static analysis alert (project, package, file, method, type, category, priority, extension, and number of alert modifications); software metrics (size, number of methods, number of classes, cyclomatic complexity); source code history (open in revision, developer, file creation version, file deletion revision); source code churn (number of added, modified, and deleted lines, growth, and percentage of modified lines); and aggregate features (total alerts for

revision, total open alerts for revision, alert lifetime, file age, alerts for artifact, and staleness).

They performed tests on two programs: `jdom` and `org.eclipse.core.runtime`. They collected source code history and code churn. They checked out every 25th revision and built the project. If the project failed to build it was skipped. They gathered the size and complexity metrics. By comparing one version to the next they were able to gather all the required ACs. For alert characteristic feature selection they used Best First, Greedy Stepwise, and RankSearch. Using Weka, they ran several classifiers using ten-fold cross validations and default settings for each method used. They evaluated each set using several default machine learning algorithms and presented their results.

The authors found that a subset of ACs should be project specific. They also found consistencies in which ACs were selected or excluded. Their results included averages for precision (89%, 98%), recall (83%,99%), and accuracy (87.8%, 96.8%) for `jdom` and `runtime` respectively.

An attempt to identify actionable static analysis alerts was presented in (Ruthruff et al., 2008). The authors created a statistical model leveraging logistic regression to classify results as true, false positives, or actionable. They tested their model on Google® source code over a three-month period in 2007.

The code factors evaluated by the model included 33 factors: the FindBug warning descriptors; their in-house tool at Google® titled BugRank in which developers rank the bugs in priority from 0-100; file characters of age and extension; history of warnings in code from file and project warnings, and file, package and product staleness; source code factors of depth, file length, and indentation; churn factors such as added, changed,

deleted, growth, and both total and percentage of lines changed. They used a screening methodology for selecting a subset of at least six predictive factors to use as independent variables for their logistic regression models. This consisted of four stages. In each stage the percentage of alerts they evaluated was increased. For each factor in each stage, they performed an analysis of deviance from the logistic regression model using a Chi-squared test and eliminated factors with small effect sizes using a gradual reduction of  $p$ -values. The logistic regression model was then fit using the remaining factors as independent variables.

They found that code churn factors were almost immediately eliminated and speculated that this was most likely due to the amount of code change that occurred daily. Factors such as their in-house ranking system and bug pattern were consistently selected. Further, the models built on screened data was, in general, at least as good as that of the models leveraging the entire warning data sets. The models were 85% accurate in predicting false positives and 70% accurate on actionable alerts.

An interesting piece of work using a more mature feature selection methodology than previously used in the literature; however, it cannot be reproduced as this was tested on proprietary source code. Bias may have been introduced in the priority ranking performed by developers and this factor was always selected in their models.

It was evidenced that the domain continued to be plagued with inaccuracies and high false positive rates when the capabilities of static analysis to detect security vulnerabilities was explored in (Goseva-Popstojanova & Perhinschi, 2015). The authors tested three well known commercial static analysis tools on the Juliet test set as well as three open-source programs: Gzip, Dovecot, and Tomcat. They measured accuracy,



recall, probability of false alarm, and G-score. None of the tools were able to detect all vulnerabilities. For the C/C++ test cases, 27% of the common weakness enumerations (CWE) were not detected by any tool, 32% were detected by a one or two tools, and 41% were detected by all three tools. Likewise, for the Java test cases 11% were not detected by any tool, 68% detected by a single or two tools, and only 25% were detected by all three tools. For both the C/C++ and Java vulnerabilities, none of the tools showed statistically significant differences in their detection rates. The mean, median, and recall for all tools was around or lower than 50%. The authors stated that this is comparable or worse than random guessing.

A survey paper (Muske & Serebrenik, 2016) presented a thorough review of the extant literature of research efforts on the processing of static analysis alerts. After performing a systematic search for peer reviewed works using a combination of keyword searches and snowballing, the authors reviewed and categorized the resulting papers into seven categories: Clustering; Ranking; Pruning; False Positive Elimination; Static and Dynamic Analysis Combination; Simplifying Inspections; and Design of Light-Weight Static Analysis Tools. For each category the authors provided a short review of a few works. This paper provides evidence of both the breadth and varied approaches regarding the processing of alert output from static analysis in peer reviewed literature.

Recent research efforts that utilized methods similar to the research effort performed herein include SA works in machine learning, feature selection, and classification of alerts.

An effort to classify alerts using machine learning algorithms was presented in (Yüksel & Sözer, 2013). They created their own dataset using thousands of alerts from a

digital TV software application. After classification, they trained and tested several machine learning algorithms using Weka. In addition to the alert characteristics generated from the SA tool, they gathered alert characteristics such as severity, alert code, lifetime, developer classification, file name, folder name, number of open alerts, total alerts, and alerts in module. They performed three studies: first, they used ten different attribute evaluator tools; second, they used the full data set to evaluate the accuracy of 34 machine learning algorithms; and third, they trained on alerts generated until the 5th run of the SA tool and classified alerts in later releases. In the first study, they found that file name, lifetime, alert code, developer classification, and severity were the most relevant characteristics for classification. In the second study, they found that random forest, random committee, and DTNB performed the best with accuracies over 83.6% and recalls over 83.6%. In the third study, they found that the average accuracy, precision and recall was around 90% on the third test set; however, the third test set had a higher number of true positives in the test set.

(Hanam et al., 2014) proposed an alert classification and ranking method by applying machine learning techniques to find patterns in the source code near the source of the alert. Their method involves backwards program slicing near the source of the alert. The source code is parsed into an abstract syntax tree (AST) which is used to build a call graph and pointer analysis. The call graph, pointer analysis, and the alert seed statements are used to construct backwards slices for each alert. They then determined alert characteristics for each statement type. To classify the alerts they used decision trees, Naive Bayes, and Bayesian network in Weka. They tested their method on FindBug alerts

from Tomcat6, Apache Log4j, and Apache Commons. They discovered alert patterns do exist and their method also improves actionable ranking.

(Yoon, Jin, & Jung, 2014) reduced the false alarm rate by classifying alerts using a support vector machine (SVM). They created an AST from the source code and then performed feature vector extraction from the AST as a preprocessing step. They then trained and tested their SVM classifier on ten open source Java applications. They were able to reduce the false positive alarms by 37.33%.

An attempt to detect and correct vulnerable code using data mining techniques was presented in (Medeiros, Neves, & Correia, 2014). Their method involved four steps: a web application protection (WAP) taint analyzer for finding vulnerable code, data mining to learn and classify false positives, code correction to resolve the vulnerable code, and feedback to present information back to the developer. For the data mining module that predicted false positives from the output of the WAP tool, they used Weka to discover which algorithms would perform best for their data. They found that logistic regression, SVM, and random tree were the top performers. They ultimately choose to implement logistic regression for their classification. They tested their model on 35 open source PHP applications. Their model resulted in an accuracy of 92.1% and precision of 92.5%.

A method using feedback to train classifiers to reduce false positives was presented in (Tripp, Pistoia, & Aravkin, 2014). This method took the raw output of static analysis alerts and asked users to classify a small subset as true or false. This information, along with selected warning attributes, were used to train different classifiers. Each of the candidate filters were applied to the test set and scored. The filter that attained the highest score was used to classify the remaining alerts. They manually choose features of: source

identifier, sink identifier, source line number, sink line number, source URL, sink URL, external objects, total results, number of steps, time, number of path conditions, number of functions, rule name, and severity. For the learning algorithms they evaluated: Naive Bayes, OneR, SVM, J48, and a Naive Bayes tree. They tested their method on security warnings from 1,706 HTML pages. In all cases the model was able to improve precision by a factor between 2.8 and 16.6 times. This model was implemented into a leading commercial SA tool.

Machine learning techniques to predict cross-project vulnerabilities in source code was explored in (Abunadi & Alenezi, 2015). The authors built fault prediction models based on two projects and ran the models on a third to measure its prediction power. They used a previously collected dataset that contained software and vulnerability information regarding three PHP open-source web applications: Drupal, Moodle, and PhpMyAdmin. Code characteristics included: lines of code, lines of non-HTML code, number of functions, cyclomatic complexity, maximum nesting complexity, Halstead's volume, total external calls, fan-in, fan-out, internal functions or methods called, external functions or methods called, and external calls to functions or methods. They applied Naive Bayes, logistic regression, support vector machine, J48, and random forest classifiers to the datasets using Weka. J48 and random forest outperformed the other classifiers. Using the two best performing models they ran predicted errors in the third project and both had high prediction rates. For both models the metrics were within hundredths of each other with around 98% precision, 96% recall, and 97% for F-measure. This work shows promise to cross-project vulnerability prediction; but it must be stated that the code bases tested are very similar.

Several previous papers include source code history in their alert processing models; however, (Hovsepyan, Scandariato, & Joosen, 2016) tried to quantify how much history is beneficial. To investigate, they used Mozilla Firefox and Google Chrome. They gathered the programs revisions and associated histories, as well as data from the Mozilla Foundation Security Advisor (MFSA), National Vulnerability Database (NVD), and Chrome Issues Tracker (CIT). For each security issue they mapped the vulnerability to the related files. The authors used two different methods for feature selection. One was 38 traditional code level metrics such as lines of code, count, cyclomatic complexity, ration comment to code, highest amount of nested conditional statements, etc. The other method was a bag-of-words approach measuring the frequencies of the tokens appearing in the source code. To test prediction of their models, for each application they selected previous versions, built that version and used that version to predict vulnerabilities in later versions. This idea was set forth by Shin (Shin & Williams, 2013) as well as Scandariato (Scandariato, Walden, Hovsepyan, & Joosen, 2014). They then ran the models to predict vulnerabilities in the next release. The ones with more history performed better but at a cost of file inspection ratio. The authors ultimately determined that recent history is more beneficial.

An effort to accurately find and correct cross-site scripting (XSS), SQLi, and other injection attacks in PHP code was attempted in (Medeiros et al., 2016). Their system was composed of three modules: Code analyzer, FP predictor, and Code Corrector. To train the FP predictor, the authors manually classified 76 vulnerabilities as either FP or TP using 15 data characteristics pulled from the WAP tool. They then used Weka to determine which classifiers performed best for their model: Logistic Regression, Random

Tree, and SVM. For prediction they used a voting methodology; however, results were not provided on how often the classifiers disagreed. They evaluated their solution on 45 open source packages of all sizes and application types and found their predictor was 92% accurate. This effort only evaluated one language for two types of vulnerabilities.

The statistical correlation between actionable alerts, unactionable alerts, and defects was explored in (Yan et al., 2017). They took 40 releases from three open source applications: MyFaces, Camel, and CXF, and tested them using the static analysis tool FindBugs. They classified the alerts as either actionable or unactionable using Heckman's (Heckman & Williams, 2009) method of classifying alerts. This method classifies an alert as actionable if it is removed from one version to another; remaining alerts are classified as unactionable. They then collected defect data from Jira reported bugs from Bugzilla. They then performed statistical calculations to determine if there was a correlation between alerts and defects; and additionally, if there was a correlation between actionable alerts and defects. They found that the overall quantity of alerts was not an indication of defects; however, they did find that actionable alerts was an indication of defects.

The application of deep neural networks to the discovery of vulnerable software components was recently published by (Pang et al., 2017). Their model used statistical n-gram analysis feature selection prior to a deep neural network (DNN) for classification. They tested their model on four Java applications and obtained averages of 92% accuracy, 95% precision, and 90% recall. These are promising results and prove that DNNs can be successfully applied to the SA domain; however, the paper lacks through details preventing the replication of their work. Although this paper is not classifying

static analysis alerts but rather predicting vulnerable software components the feature sets may be similar.

Predicting false positive alerts using program slicing to learn program structures that cause false reports was presented in (Koc et al., 2017). Their method first involves code reduction by taking the body of the method where the alert was generated from as well as a backward slice from the warning line. They used this reduced code to train two classifiers: a Naive Bayes and a long short-term memory (LSTM) classifier. They tested their models on the OWASP benchmark test suite. The LSTM classifiers performed better than the Naive Bayes. The LSTM using method body had 81% recall and 89.6% accuracy while the LSTM using backward slicing had 97% recall and 85% accuracy.

It is evidenced that classifying static analysis alerts is still an active problem in the research community. Recent efforts to address the problem have included data mining, feature selection, and machine learning. These methods have shown promise; however, more work is still needed.

## **Chapter 3**

### **Methodology**

#### **Introduction**

To address the problem of high false positive rates the goal of this research effort was to develop and evaluate methods for feature selections that helps to improve the classification accuracy of static analysis alerts. This research effort presented and tested a novel method leveraging feature selection that resulted in the improved classification of alerts.

A feature selection method was developed and evaluated to investigate the improved classification accuracy of static analysis alerts. After data was gathered and preprocessed, the data was split into train and test sets. A genetic feature selection model was trained and tested on the train and test sets respectively. The process was performed iteratively, testing selected feature subsets for an improvement in classification accuracy in an embedded fashion. This process resulted in a subset of relevant features for the classification of the alerts. To quantify the feature selection model's classification improvement, the model was compared with a control classifier that excluded the feature selection component.

#### **The Model**

The model first performs feature selection. Second, classification is performed. Third, data analysis is performed.



### *Feature Selection Method*

Feature selection methodologies were developed and evaluated for the improved classification accuracy of static analysis alerts. After data was prepared and pre-processed, the first step in the model was to perform feature selection.

For the feature selection component genetic algorithms were employed. Genetic algorithms (GA) are based upon Darwin's theory of evolution. They are used to find optimal solutions to difficult problems, for instance an optimal set or optimal shape. GAs utilize an objective function, called a fitness function, to progressively evaluate individuals. Better performing individuals of each generation are selected for breeding. The GA performs crossovers, mutations, and selection of the fittest to arrive at an optimal generation referenced as a solution (Holland, 1975).

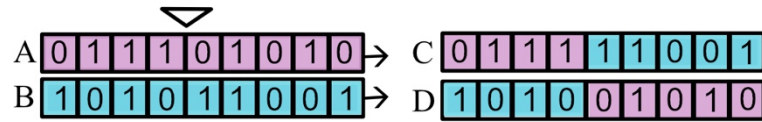
The general outline of a GA is:

```
Create the initial population of chromosomes.
For each of N generations {
    Selection: Select parents based on the fitness, with replacement.
    Recombination: Pair parents and perform recombination to produce
                   offspring.
    Mutation: Mutate offspring.
    Replacement: Replace the parents with the mutated offspring.
}
```

GAs are iterative functions that begin with an initial population. A population is a subset of all the possible solutions. Chromosomes represent one solution. A gene represents one element in the chromosome. In each generation, chromosomes are evaluated on their performance on the fitness function. Better performing chromosomes have a higher probability to mate and, thereby create even better fit chromosomes.

Crossover is used to create new chromosomes from the existing better performing chromosomes. The point at which each chromosome is separated is called the crossover

point. Crossover points may be random or predefined. There may also be single or multiple crossover points. In the case of a single crossover point, the first section from each chromosome will be merged with the secondary section of the other chromosome. Figure 1 shows an example of a single point cross over. This is a simplistic example of crossover; however, several methods to perform crossover functions exist including uniform, sequential constructive (SCX), position based (POS), partially mapped (PMX), order-based (OX1), and more. This overall crossover process results in offspring that is composed of genes from both sets of the parent chromosomes.



*Figure 1* GA Crossover

Mutation, a random inversion or minor modification to a gene, is performed to ensure diversity in the generations. This helps to prevent the algorithm from getting stuck in a local minima by exploring the search space. Mutations may flip bits, perform string manipulation, swap values, invert subsets, or other types of random minor modifications.

This overall process continues until a termination condition is met. Termination conditions may vary but include a predefined number of generations, a sufficiently performing solution presented, or a plateau of consecutive generations performance on the objective function.

There are different methods of feature selection including filter, wrapper, embedded, and more recently hybrid (Li et al., 2018). Filter methods do not consider learning algorithms but use statistical measures to determine feature importance. Features are

ranked and either kept or removed from the dataset. They are more computationally efficient than wrapper methods but feature sets may not be optimal for predictive models. Wrapper methods take the selected feature sets and evaluate their accuracy on a predictive model. Each subset is used to train a model and then tested. The classification accuracy is used to score the performance of the subset. This can be computationally expensive but often provides the best performing feature set for the model. However, the selected feature set may not generalize to other data as this method is prone to overfitting. Embedded methods perform feature selection and classification simultaneously. It is similar to wrappers in that it considers the predictive model's performance; yet it is less computationally expensive and less prone to overfitting. Hybrid models are some sort of combined method of filter, wrapper, and/or embedded (Li et al., 2018).

The following outlines the employed genetic algorithm specifics.

**Method:** The method used for the feature selection model was an embedded model.

**Representation:** Binary encoding was used to represent feature selection or exclusion in the solution set. Each chromosome, candidate solution, was represented as a bit string of length  $n$ , where  $n$  was the total number of features. The  $j^{th}$  feature was retained if the  $j^{th}$  bit was 1 and removed if the  $j^{th}$  bit was 0.

**Initial Population:** The initial population was randomly selected. This was performed by randomly generating bit strings of length  $n$  as members of the initial population. The probability that any bit in a chromosome was a 1 bit was independently 0.5.

**Fitness:** The fitness of a chromosome was proportional to the classification accuracy of the model on the test set using the selected subset of features.

**Selection:** Tournament selection was used. Selected chromosomes (parents) from the current population were placed in a mating pool. Selected chromosomes randomly mated using recombination to create offspring.

**Recombination:** A standard single-point crossover was used. With probability  $p_c$ , the crossover operation was applied, and with probability  $(1 - p_c)$  the offspring were identical to the parents.

**Mutation:** With a small probability  $p_b$ , a random bit in a chromosome was inverted; with probability  $(1 - p_b)$  the chromosome remained unchanged.

**Replacement:** The offspring generated through recombination and mutation replaced the parents in each generation. Elitist replacement strategies were used.

**Termination:** Generations continued to be created and evaluated until improvement of the fitness function was absent, minimal for a number of generations, or until a predefined number of generations were evaluated, whichever occurred first.

**Parameters:** The GA was run with several combinations of the settings as fully outlined in Appendix C. These included variations on population sizes, generations, selection rates, mutation probabilities, and termination conditions.

**Code:** The complete genetic algorithm code is included in Appendix I.

### *Classification Method*

The classification method used for this research was a support vector machine (SVM), a machine learning classifier. For both the control classifier and the feature selection model a SciPy Linear SVC with default settings was used.

An SVM is a supervised method for classifying objects introduced in 1963 by Vladimir Vapnik. He later worked with Alexey Chervonenkis to refine the algorithm. The

algorithm classifies objects by finding an optimal hyperplane, or decision line, that distinctly separates objects in the data set (*Fig. 3  $H_0$* ). Figure 2 shows several possible hyperplanes that separates the two classes of objects.  $H_1$  has some classification errors,  $H_2$  and  $H_3$  correctly classify all objects; however, the distance between the two classes is greater in  $H_2$ . Thus, SVMs not only want to separate the classes but separate them as distinctly (i.e., optimally) as possible. The objective becomes to maximize the margin, which is the area between the positive and negative hyperplanes. Larger margins have lower generalization errors whereas smaller margins are more prone to overfitting. Once the hyperplane is determined, classification of new objects occur based upon which side of the hyperplane the object falls. The hyperplane could be linear or non-linear. Finding the hyperplane for non-linearly separable data can be accomplished by using a kernel trick, projecting the data into a higher dimensional feature space. The non-linear hyperplane may then be found. Pushing the data and hyperplane back onto the original feature space, the hyperplane appears to weave through the data set. Additionally, SVMs have a single global minimum (Kowalczyk, 2017; Russell & Norvig, 2014).

The equation for a hyperplane is:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

This is another way of writing the two-dimensional equation of a line  $y = ax + b$ . However, by using vectors it also works for finding a hyperplane in multi-dimensions.

Items are classified depending on which side of the hyperplane they fall. Relative to the decision line, positive items (1) reside further than the positive hyperplane (*Fig. 3*

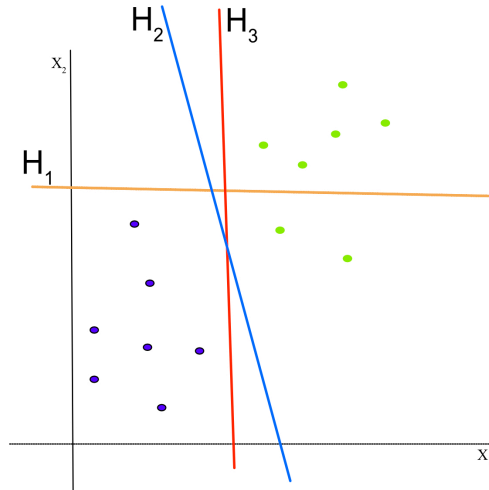


Figure 2 Possible Hyperplanes

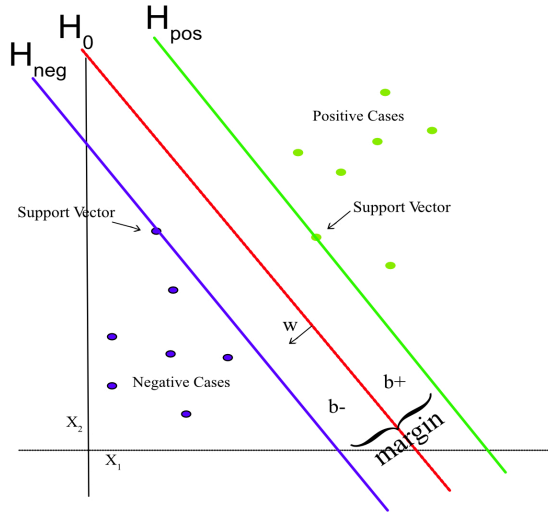


Figure 3 Linear SVM Hyperplane

$H_{pos}$ ) and negative items (-1) reside further than the negative hyperplane (Fig. 3  $H_{neg}$ ).

Support vectors reside on the hyperplanes. The positive and negative hyperplanes are defined by:

$$\begin{aligned}\mathbf{w} \cdot \mathbf{x}_i + b &= 1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &= -1\end{aligned}$$

Which is a dot product of the vector normal to the hyperplane  $\mathbf{w}$  and the vector  $\mathbf{x}_i$  plus the bias  $b$ . By using these two hyperplanes the margin can be computed. The margin is defined as:

$$m = \frac{2}{\|\mathbf{w}\|}$$

Classification of items use the following with a constraint to ensure that no data point resides inside the margin.

$$\begin{aligned}\mathbf{w} \cdot \mathbf{x}_i + b &\geq 1, \text{ if } y_i = 1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &\leq -1, \text{ if } y_i = -1\end{aligned}$$

Which can be combined to:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \forall_i$$

SVMs want to maximize the margin and thus minimize the norm or  $\|\mathbf{w}\|$ . Rather than maximizing the margin it is easier to minimize  $\|\mathbf{w}\|$  which becomes the constrained optimization problem:

$$\begin{aligned} & \underset{(w,b)}{\text{minimize}} && \frac{1}{2} \|\mathbf{w}\|^2 \\ & \text{subject to} && y_i(\mathbf{w} \cdot \mathbf{x}_i) + b - 1 \geq 0 \\ & && i = 1, 2, \dots, m \end{aligned}$$

This is a quadratic problem which can be solved using the Lagrangian multiplier method resulting in the following SVM primal optimization function:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^m \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]$$

The above problem can be solved by taking the Wolfe dual of the above primal problem:

$$\begin{aligned} & \underset{\alpha}{\text{maximize}} && \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ & \text{subject to} && \alpha_i \geq 0, \text{ for any } i = 1, \dots, m \\ & && \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned}$$

Which has removed the dependence on  $w$  and  $b$ .

By satisfying the Karush-Kuhn-Tucker (KKT) condition, the problem can be solved by computing just the inner products of  $x_i, x_j$  while also guaranteeing the optimal solution.

The following SVM classification hypothesis is derived:

$$h(\mathbf{x}_i) = \text{sign}\left(\sum_{j=1}^s \alpha_j y_j (\mathbf{x}_j \cdot \mathbf{x}_i) + b\right)$$

Soft margin SVMs allow for noisy data creating outliers that could alter margin calculations. Slack variables were introduced to relax the constraints, thereby allowing for some classification mistakes. The goal is not to have zero misclassifications but rather only a few and penalize any classification errors. Thus, the new constraint becomes:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \zeta_i$$

To limit the number of incorrect guesses the sum value of all  $\zeta$  must be minimized. Additionally, the sum must not be negative. A new conditional is added to the objective function becoming:

$$\begin{aligned} & \underset{\alpha}{\text{maximize}} && \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ & \text{subject to} && 0 \leq \alpha_i \leq C, \text{ for any } i = 1, \dots, m \\ & && \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned}$$

Choosing a small  $C$  will give a wider margin and more classification errors. The alternative it true that a larger  $C$  will give a harder margin with less errors.

Kernel functions return the dot product as if they had been transformed into vectors without actually transforming them. This minimizes computation effort. By adding the kernel function  $K$ , the dual problem becomes:

$$\begin{aligned} & \underset{\alpha}{\text{maximize}} && \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i \cdot \mathbf{x}_j) \\ & \text{subject to} && 0 \leq \alpha_i \leq C, \text{ for any } i = 1, \dots, m \\ & && \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned}$$

The hypothesis therefore becomes:



$$h(\mathbf{x}_i) = \text{sign}\left(\sum_{j=1}^s \alpha_j y_j K(\mathbf{x}_j \cdot \mathbf{x}_i) + b\right)$$

There are several kernel functions. Some commonly used kernel functions include:

- Linear  $K(\mathbf{x}, \mathbf{x}') = \mathbf{x} \cdot \mathbf{x}'$
- Polynomial  $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + c)^d$
- RBF / Gaussian  $K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$

SVMs have previously been used to classify static analysis alerts in (Bleier, 2017; Medeiros et al., 2016; Tripp et al., 2014; Yi, Choi, Kim, & Kim, 2007; Yoon et al., 2014).

For this research, both the control classifier and the feature selection model leveraged a SciPy SVM Linear SVC with default settings. The classification confusion matrix is presented in Table 1.

*Table 1* Confusion Matrix

		<i>Actual</i>	
		<i>Positive</i>	<i>Negative</i>
<i>Classified</i>	<i>Positive</i>	True Positive (TP)	False Positive (FP)
	<i>Negative</i>	False Negative (FN)	True Negative (TN)

## Measures

Accuracy, precision, recall, F-measure, and the false positive rate, were used to evaluate the model's performance. Table 2 details the metrics used and the directions that indicate improved classification.

*Table 2* Definitions and Metrics

Name	Formula/Notation	Improvement	Description
True Positive	<i>TP</i>	Increase	The alert is true and classified correctly.
True Negative	<i>TN</i>	Increase	The alert is false and classified correctly.

False Positive	$FP$	Decrease	The alert is false and classified incorrectly as true.
False Negative	$FN$	Decrease	The alert is true and classified incorrectly as false.
False Positive Rate	$\frac{FP}{FP + TN}$	Decrease	The proportion of negative instances incorrectly classified as positives.
False Negative Rate	$\frac{FN}{FN + TP}$	Decrease	The proportion of positive instances incorrectly classified as negatives.
Accuracy	$\frac{TP + TN}{TP + TN + FP + FN}$	Increase	The proportion of correctly classified instances, either true positives (TP) or true negatives (TN)
Precision	$\frac{TP}{TP + FP}$	Increase	The proportion of true positives classified correctly.
Recall	$\frac{TP}{TP + FN}$	Increase	Also referred to as the true positive rate or sensitivity, is the proportion of true positives correctly classified as positives.
F-Measure	$\frac{2 * TP}{2 * TP + FN + FP}$	Increase	The harmonic mean of precision and recall.

## Data Sets

Data sets in the static analysis domain are limited (Herter et al., 2017; Heckman & Williams, 2008; Shiraishi et al., 2015). As a result, researchers in this domain often generate their own data sets using a predefined methodology. A thorough review of the existing data sets was performed and compared with the data set requirements for this research. Although some data sets were promising, upon further investigation each lacked at least one necessary component. However, one was found to contain most of the required elements. Therefore, that data set was utilized; however, it was augmented with the additional static analysis components needed. The additional components were gathered by following the framework as outlined in the literature (Heckman & Williams, 2008, 2009).

The data set criteria for this effort was as follows. The data set must contain real world labeled static analysis alerts. A problem with artificially generated data sets is that they are not a true representation of real-world data. Real world applications contain bona fide developer errors, complex variable paths that are difficult to follow, and often contain multiple flaws per function. Additionally, the source code of the applications must be available. Moreover, the data set should contain source code metrics (such as code churn, fan-in, fan-out, etc.) and historical data (such as the lifetime of alerts and the types of alerts resolved). If they are not included, they must be easily calculated given the other information provided in the data set. Furthermore, the data set must contain a sufficient number of test cases. Ideally, the data set contains ample alerts regarding software security issues and not just software bugs. Finally, if the data set does not contain SA alerts, the application must have several versions, issue tracking systems, and published vulnerabilities in order to generate and label the alerts.

Static analysis test suites are designed to create data sets for tool comparisons. These test suites are collections of source code with labeled good and bad test cases. The data sets are generated by running the test cases through the SA tools and labeling the alerts by matching them to the known list of good and bad test cases. Several test suites exist for static analysis; however, they lacked the required features.

### *Test Suite and Data Set Evaluations*

JULIET, Benchmark and the Software Assurance Reference Dataset (SARD) are labeled vulnerable software test suites that have previously been used in the literature to generate static analysis alerts. These test suites were specifically designed to test and study static analysis tools. The test suites consist of functions that contain intentional

security vulnerabilities or programming flaws. Each function is designed to test for one issue and is mapped to its related MITRE's CWE. Test cases also contain functions in which no known flaw exists. However, the creators do note that in some instances other unrelated flaws may also be present. The test cases are packaged by CWE and therefore can be tested individually or in concert using static analysis tools. The resulting alerts can be easily labeled.

The JULIET Test Suite was created by the National Security Agency's (NSA) Center for Assured Software (CAS) to evaluate static analysis tools (NIST, 2017a). Test suites for both Java and C/C++ are provided. The Java version 1.3 contains 28,886 test cases covering 112 CWEs in more than 46,000 files using over 4 million lines of code. The C/C++ version 1.3 contains 64,099 test cases and over 100 classes of errors in more than 100,000 files using over 8 million lines of code. The set consists of buildable code files labeled in a systematic method. Each test case contains one type of test for the flaw in a function labeled as 'bad'. Additionally, there are also test cases in the same file labeled with some inclusion of the string 'good' (ie. good, goodG2B, goodB2G, good1, etc.). Helper methods are labeled containing some string of 'helperBad' or 'helperGood' indicating that it is a helper function to the ultimately 'good' or 'bad' function. Sources and sink methods are also labeled with some string containing 'badSource', 'badSink', 'good\*Source', 'good\*Sink'. Additionally, the naming convention of the test case files includes the CWE and test number. Although this test suite has the potential to create an adequate number of labeled static analysis alerts, it lacks historical features, source code metrics, and is not reflective of real-world source code. Additionally, although the

labeling of alerts is easily performed based on the naming conventions, the naming conventions alone could be picked up by scanning tools as patterns.

The OWASP Benchmark project was created to test and compare static analysis tools. Version 1.2 was released in June of 2016 and consists of 2,740 test cases covering 11 CWEs and is a complete web application with a UI such that test cases are fully exploitable. The suite contains an expected results CSV file that labels test cases as true or false and maps them to the related CWE. All test cases reside in the folder 'testcode' and the naming convention of the files are generic. Functions that reside in test cases have generic names such as 'doGet', 'doPost', etc. Each test case is a servlet or JSP and is either a true positive or a false positive test case. The test suite also consists of a scoring portion in which scan results from tools may be imported and then automatically ranked for the comparison to other tools. The scoring outputs the true positive rate, false positive rate, true negative rate, and it's Youden Index. This test suite also has the potential to create an adequate number of labeled static analysis alerts. It uses an external list and not the source code itself to label the test cases. However, it lacks historical features and realistic source code metrics.

The ability to create labeled data sets is a clear benefit to using these artificial test suites as all true and false positives are known. They also offer excellent opportunities to compare static analysis tools; however, the comparison of static analysis tools was not the goal of this research effort.

Limitations are inherent as they are artificial in nature and are not reflective of natural code bases. They lack the complexity of natural code as the test cases have been reduced

to their simplest form in order to test for one issue per test case. As a result, the alerts generated from the tools may not reflect real world alerts.

Another limitation to these test suites is the frequency of flaws included in the suites. Some flaws may present more often than others. Therefore, the test cases and subsequent alerts could be skewed. Another limitation is that several features that may assist in alert classification are absent such as source code metrics and historical data.

It is also noteworthy that these test suites are available to tool vendors. Vendors may use these test suites to improve their scanning techniques and thereby improve their accuracy on these tests. This is a clear advantage for the vendors of these tools as they possess the answers to the benchmark tests. Therefore, these test suites were not sufficient for this use case.

Several other data sets were reviewed and compared to the pre-defined criteria. A matrix of the findings is presented in Table 3.

*Table 3 Data Set / Test Suite Requirements Matrix*

Name	Source	Test Suite or Data Set	Source Code	SA Alerts	Code Metrics	Historical	Security	Realistic
JULIET	NIST, 2017a	Test	✓				✓	
Benchmark	OWASP, 2017a	Test	✓				✓	
WebGoat	OWASP, 2017b	Test	✓				✓	
Toyota ITC	Shiraishi et al., 2015	Test	✓					
Software Defect Prediction Set	Mausa, Grbac,&Basic, 2014	Data	✓		✓	✓		✓
Bug Prediction Set	D'Ambros, Lanza, & Robbes, 2012	Data	✓		✓	✓		
FaultBench	Heckman & Williams, 2008	Data	*	✓	✓	✓		✓
PHP Security Data Set	Walden, Stuckman, & Scandariato, 2014	Data	✓		✓	✓	✓	✓

*\*Source code available. Evidence of errors building packages again in subsequent works (Bleier, 2017).*

### *The Selected Data Set*

A public data set containing security vulnerability data and machine learning features of three open source PHP applications has recently been cited in the literature referred to as the PHP Security Vulnerability Dataset (Walden, Stuckman, & Scandariato, 2014).

The complete raw data set, replication data set, and all scripts used to create the data sets can be downloaded from <https://seam.cs.umd.edu/webvuldata>. The data set contains 233 verified security vulnerabilities and has been used for subsequent studies in (Abunadi & Alenezi, 2015; Walden et al., 2014; Zhang et al., 2016). The authors collected data from 95 versions of PhpMyAdmin from 2.2.0 and 4.0.9, 71 versions of Moodle from 1.0.0 to 2.6.1, and 1 version of Drupal v6.0.0.

PhpMyAdmin is a web-based database administration tool for MySQL initially released in 1998. Moodle is an online learning management system first released in 2002. Drupal is a web content management system initially released in 2000. All of these applications have ample release history, change history, and published security vulnerability information.

The authors collected the source code and release history of each version. All three applications used Git to house their repositories, thus the authors were able to download the main branch which included previous release information. Included in the data set is a file for each applications version, Git hash for the release, and dates for the release.

Source code metrics were collected and included in the data set. Metrics were included and linked to file names for each project and version. These include: lines of code, lines of non-html code, number of functions, cyclomatic complexity, maximum nesting

complexity, Halstead's volume, total external calls, fan-in, fan-out, internal functions called, external functions called, and external calls to functions.

The authors collected vulnerability information for each project and version by gathering data from the NVD and security announcements from the product. Included in the data set is the Git hash of the version the vulnerability was introduced, the Git hash of the version in which the vulnerability was resolved, the associated CVE identifier, and the file associated with the fix.

They tracked vulnerabilities throughout versions for PhpMyAdmin and Moodle. They did not use multiple versions of Drupal; thus, vulnerability tracking information was not performed for Drupal. The data set includes a matrix of the file associated with each vulnerability tracked over versions.

They compiled tokens of the source code for text mining. They parsed through the source code files, extracted the PHP tokens, and then labeled the resulting concatenated tokenized string as vulnerable or not vulnerable.

They merged the data together and evaluated the data set using machine learning to predict defects. They published their scripts for study replication, source code of the applications, and the collected features as R and Weka files.

A limitation of this data set is that this only covers three PHP applications, the authors excluded several published vulnerabilities, and the set does not include labeled static analysis alerts. However, of all the data sets evaluated, this data set was the closest to the desired criteria. These releases and their related source codes were still available on each applications website archive for download. Additionally, all of these applications bug tracking systems were available for query. Therefore, this data set was leveraged;



however, augmented with labeled static analysis alerts using the framework as outlined in (Heckman & Williams, 2008, 2009).

For this research effort, the Drupal versions were expanded to include 38 versions from 6.0.0 to 6.38. The PhpMyAdmin and Moodle versions evaluated remained unchanged. Each release of each application was scanned by each tool and the resulting alerts labeled. The existing data set contained 233 known TPs and the resulting static analysis alerts from the tools exceeded 250,000. As a result, the existing data set was not sufficient to label the static analysis alerts. Additional work was performed to match auxiliary information such as change logs, security notices, release notes, bug tracking systems, and CVEs to the alerts. For labeling, alerts were tracked between versions and auxiliary information inspected. Additional software metrics were gathered and merged to augment this data set to create a static analysis data set.

#### *Overall Framework for Gathering Alerts*

The method for creating and labeling static analysis alerts was well outlined in (Heckman & Williams, 2008, 2009). The overall framework consists of four steps.

1. Generate subject revision history: source code is gathered with versioning and change history.
2. Build Process: if required, build the version. Compute code metrics, run through static analysis, and gather the alerts.
3. Alert Classification: use source code histories to track and label the alerts throughout versions.
4. Artifact Characteristic Generation: gather information about the alerts and surrounding source code that may be predictive factors.

This overall process was followed with some additions. In addition, the CVEs, release notes, security notices, and change logs were gathered, and the bug report systems queried. This additional information was used to assist in alert labeling and feature generation.

### *Detailed Framework for Gathering Alerts*

Scanning the source code to create static analysis alerts was a trivial matter as all source code was downloaded and utilized a Git repository. A script was written to iteratively checkout each projects version, send a request to each scanner to scan the version, query the scanner for the results, and download the scan results. Output features and formats varied between scanners. Formats included CSV, JSON, TXT, XML, and proprietary vendor formats. All alert features provided by the tool were exported. To convert the results into a consistent CSV format, a script was written to cipher through the scan results and merge the alerts to allow for further processing and analysis. Additionally, alert characteristics were gathered during this process such as alert lifetimes, alert start and end versions, number of path hops, and path file names.

Two industry leading commercial scanners were used for code scanning. Permission to disclose the commercial tools names was not granted. Therefore, they are referenced as Tool A and Tool B. To protect incidental disclosure, identifiable information regarding these tools will not be disclosed such as unique identifiers or features specific to those tools.

In addition to the commercial scanners, an open source scanner was also used. SonarQube is an open source scanner capable of scanning 25 languages, including PHP. It may be downloaded and run locally or may be used as a service online. It is capable of

finding bugs and security vulnerabilities. This scanner was downloaded and installed using the default configurations. This scanner had a limitation upon exporting of scan results to a maximum of 10,000 records per API query. To overcome this obstacle, queries were written to export different alert types per query if the quantity of alerts exceeded the maximum allowed.

Upon completion of each project and version being scanned by each scanner, and the results downloaded and parsed into a consistent CSV format, the resulting features from all tools were manually evaluated. If features were not represented by all scanners or leaked identifying tool information, the feature was removed. If these features remained in the final data set, the learning algorithms could possibly learn on a particular tool rather than the alert characteristics.

All alerts were initially labeled as unactionable, or FP. By gathering multiple versions of the same application, alerts that disappear from one version to another could indicate that a bug or flaw was fixed. It could also indicate the deletion, renaming, or movement of a file. However, tracking the renaming of files is a difficult task beyond the scope of this research effort. It is possible that if a simple rename of the file was performed, a similar alert should appear in the immediate version after the disappearance of the original alert with a different file name. In this simple instance, the renamed file could be determined, and the alert matched. However, file renames beyond this simple case were considered deleted. Therefore, a Python script was written to track alerts throughout the lifetime of the project. Alerts that moved due to simple file rename cases remained unchanged. Alerts that were found to get resolved were marked as actionable, or TPs. Alerts where the originating alert files were absent in the next version and a matching

alert was not found, the alert was marked as deleted and subsequently not included in the data set.

Additional information was gathered to assist in labeling the alerts generated from the static analysis scans. Scripts were written to assist in matching the alerts to the relevant records. Labeling was performed if alerts could be linked to bug reports, change logs, security notices, release notes, issue tracking system, or vulnerability publication list. The projects were not consistent in what auxiliary information was provided; however, all projects had public vulnerability publication lists and some type of release notes or bug tracking system.

Publicly posted vulnerability information for each application was gathered using the National Vulnerability Database (NVD) by NIST at [nvd.nist.gov](http://nvd.nist.gov) and MITREs CVE Online Database at [cve.mitre.org](http://cve.mitre.org). CVEs found in one database will often be duplicated in the other database; however, it is possible one database contains vulnerabilities that another database lacks. Therefore, these databases were manually searched for each application. All necessary information regarding the vulnerability was contained in the published record including CVE, versions affected, exploit information, links to commit records and fix information, severity rating and scores, and vulnerability type. The original data set contained 233 CVEs; however, during this process 630 CVEs were discovered. This could be the result of additional discoveries over time or the addition of Drupal versions. To match alerts to the CVE records required the fix file name and line of code. To determine the files and lines of code for the fix required manual work. Although many records had links to commits and fix files, the method for linking, publicizing, and outlining the fixes were not consistent enough to compose a meaningful and reliable

script. Therefore, for each CVE, the fix file(s) and line of code(s) was/were manually determined via the CVE records and logged into a CSV. Next, a script was written to match the alerts to the CVE records by matching the fix file names, lines of code, and versions. The original data set fix files were also verified during this process and lines of code added to those records. Matched alerts were labeled as actionable, the CVE Boolean feature marked as true, and CVE identifier logged. Other information gathered but not utilized during this step included CVE severity, base, exploit, and impact scores, published dates, descriptions, start and fix versions and commit hashes, if provided.

Bug tracking systems may be used to track bugs and their resolutions. These systems keep records of the type of bug or flaw, what the issue was, how it was resolved, who fixed it and when. Some open source applications make their bug tracking systems public. The projects used in this data set had publicly facing bug tracking systems. This information was queried to find bugs that had been resolved and link them to alerts. They were linked via file names and line of code. Bug reports often do not follow a standard format and critical information may be in natural language. Therefore, the analysis of the bug reports and linking to alerts was manually performed. Matched alerts were labeled as actionable.

Source code change logs, commit histories, security notices, and release notes contain valuable information that can be linked to alerts. Commit histories specify file names, line of code, what exactly changed, and when. Change logs and release notes may specify files that changed, the reason for the change, and in what version it occurred. Security notices and release notes specify what changed and why. For each project and version, the available logs were manually reviewed. Any corresponding alerts were labeled as

actionable. It was found that the security notices and release notes often referenced specific CVEs. In those cases, the matching alerts were already labeled as actionable.

Additional features were gathered to complement the alert data set. The original data set contained some source code metrics on the file level; however, several other metrics could be gathered. Software metrics were easily gathered using Understand, Git, and basic file information.

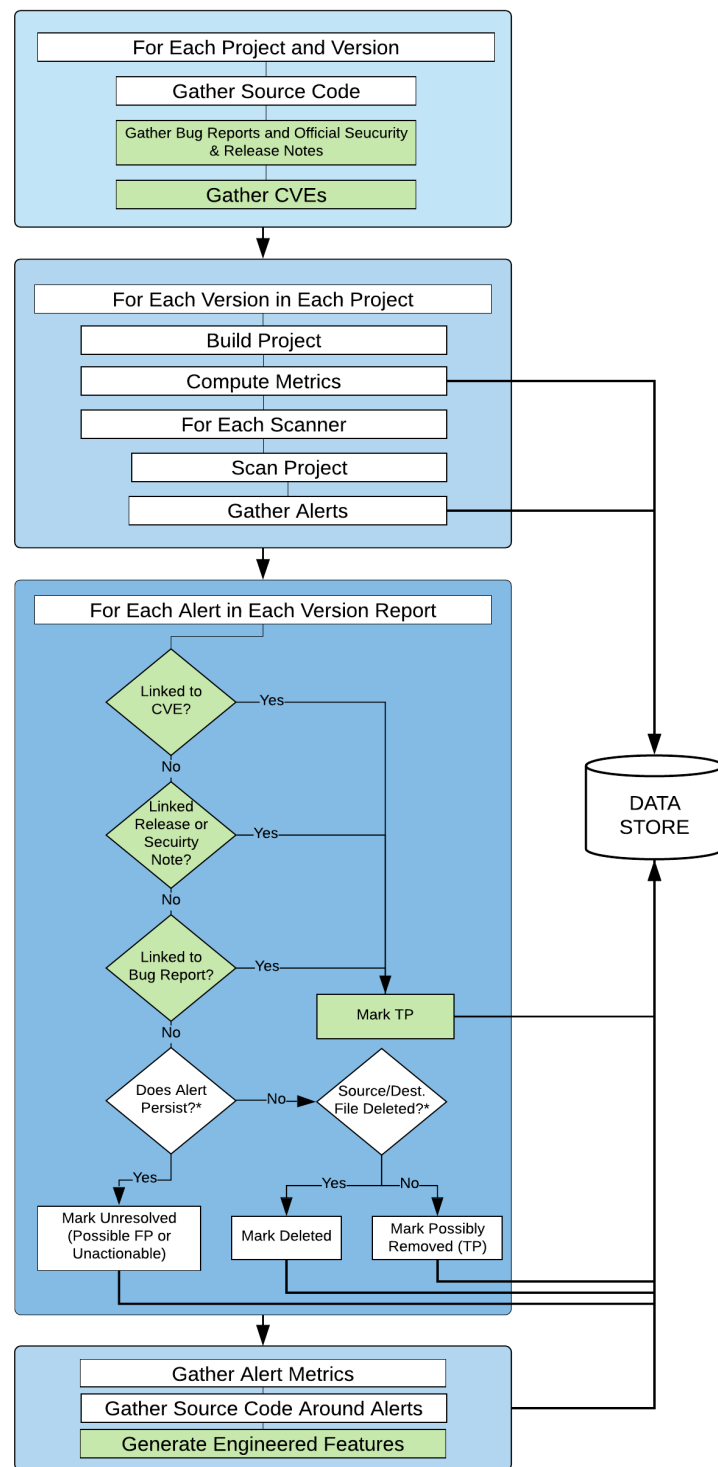
A shell script was written to iteratively checkout each project and version and then export a metrics report from the software metrics tool Understand. This resulted in a software metrics report for each project and version. It included 50 metrics and was exported into CSV's and text formats. The resulting metrics were then matched to alerts by source and destination file names and versions. The metrics were added directly to the alert for the specified files.

The original data sets file level metrics were matched to each alert using a custom script for both the source and destination files. Additional metrics were also added by matching alerts to Git commit information such as last edited date, commit dates, and authors.

The original framework for gathering and labeling static analysis alerts has now been augmented and is outlined below. A graphical workflow of this new process is displayed in Figure 4.

1. Generate Subject Revision History
  - a. Gather and verify all source code and histories.
  - b. Gather CVEs.
  - c. Gather of bug report systems, change logs, security notices, and release notes.
2. Build Process

- a. Compute source code metrics.
  - b. Iteratively for each project and each version, build the project then run the source code through the specified static analysis tools.
  - c. Download the alerts from the tools as CSV, XML, JSON or other standardized tool output.
3. Alert Classification
- a. Using alert and source code histories, label the alerts following the labeling process.
  - b. Label alerts if matched to CVEs.
  - c. Label alerts if matched to bug report systems, change logs, security notices, and release notes.
4. Artifact Characteristic Generation
- a. Gather metrics regarding alert lifetimes, resolution type, and other historic alert features. Match to alerts as features.
  - b. If possible, gather source code around the alert.
  - c. Match additionally gathered or engineered metrics and features to alerts.



\* Simplified Algorithm. Actual algorithm performs additional verifications.  
 \*\* Green Boxes are New to the Framework Alert Labeling Process

Figure 4 Framework for Static Analysis Alert Generation and Labeling



### *The Resulting Data Set*

The result of this process was a data set for each project containing labeled real-world static analysis alerts from three static analysis scanners, complete with source code metrics and historical features of the alerts. The data sets altogether included 256,198 alerts. Alerts labeled as deleted were excluded from the experiments bringing the number of alerts to 207,259.

The resulting data set is a mixture of the original data sets information merged with labeled static analysis alerts with additional features and metrics. A complete feature list and the feature's origination is presented in Appendix A. The statistics of the raw project data sets are presented in Table 4. There resulted a data set for each project, each version, as well as a combined data set (a compilation of all the data sets).

*Table 4 Raw Data Set Alert Statistics*

	Total Raw Alerts	Actionable (TP)	Unactionable (FP)	Deleted
Drupal	3,834	491	3,343	0
Moodle	126,427	67,345	39,509	19,573
PhpMyAdmin	125,937	75,960	20,611	29,366
Total	256,198	143,796	63,463	48,939

## **Experiments**

Experiments were performed to train and evaluate the feature selection model. Python was used to create and evaluate the model. The model's selected feature subsets and classification performance metrics were output for each experiment. To measure improved classification accuracy, the feature selection model was compared with a similar model that excluded the feature selection component. Therefore, for each experiment, a SVM classifier was trained and tested on the same train and test data sets and compared to the feature selection model's performance.

### *Data Pre-processing*

Prior to running experiments, the respective raw data sets were pre-processed which included the removal of duplicate records, changing categorical features into contiguous, normalizing ranges, and ensuring adequate representation of features and alert types. After pre-processing, the data sets were split into train and test sets. Once train and test sets were created, the experiments were performed to evaluate the feature selection model's performance.

Several pre-processing steps were performed on the raw data sets to prepare the data for machine learning. Python scripts were written to perform the preprocessing steps using Pandas, Numpy, and SciPy functions. For each raw data set for each experiment the following was performed. All alerts labeled as 'deleted' were removed. Duplicate alerts were also removed. Alerts were considered duplicates if the following feature values were identical: project, tool, priority, category, type, code/bug/vuln, language, CWE id, OWASP 2013 Boolean, OWASP 2017 Boolean, OWASP Top Ten 2013, OWASP Top Ten 2017, source and destination file, source and destination line, source and destination column, source and destination function. Most data sets were heavily skewed between actionable and unactionable alerts. To adjust for this skewness, alerts were randomly dropped to create an equal proportion of actionable and unactionable alerts. Features that had no data at all were dropped and logged. Categorical data was capitalized, standardized, and then one-hot encoding was performed. Statistical analysis of the numerical features mean, median, modes, and standard deviations was performed. It was discovered that the mean was not an appropriate value to use for any of the numerical features. Therefore, for numerical data, missing values were replaced with the median or

mode, depending on the analysis of the feature statistics. Finally, features were dropped if all the values in the data set were identical. All dropped features for each data set were logged for later use to prevent those features from being selected in the feature selection component of the model. Similarly, new feature names created during categorical encoding were mapped to the original feature name allowing for mapping during the feature selection component. A list of dropped features from each data set is outlined in Appendix D. The resulting data set statistics are presented in Table 5.

*Table 5 Pre-Processed Data Set Alert Statistics*

	Alerts	Actionable (TP)	Unactionable (FP)
Drupal	960	480	480
Moodle	77,996	38,998	38,998
PhpMyAdmin	40,986	20,493	20,493
Total	119,942	59,971	59,971

Next, the respective data sets were split into train and test sets for each experiment. There were 200 features available prior to one-hot encoding and thousands of iterations for each experiment. The feature space complexity could create excessive run times for the experiments. It was determined during preliminary tests to further reduce the larger data sets to a more manageable quantity. During train and test splits, if the alerts exceeded 30,000 then an equal number of actionable and unactionable alerts were randomly dropped. The resulting alert counts are presented in Table 6.

*Table 6 Final Data Set Alert Statistics*

	Alerts	Actionable (TP)	Unactionable (FP)
Drupal	960	480	480
Moodle	23,398	11,699	11,699
PhpMyAdmin	12,296	6,148	6148

The final pre-processing step was to split the sets into train and test sets using a 66-33 split. 10-Fold cross validation was tested on a few SVMs. The classification accuracy was similar to the 66-33 split results; however, it increased processing time.

### *Model Creation*

The control classifier, Model A, creation was easily performed. The following was performed for each experiment. All features from the data set were included for the control classifier. Feature scaling was performed using SciPy's Robust Scaler function with default settings. Preliminary analysis of several SVM kernels was performed on the data set and it was determined, based upon processing times and overall accuracies, that the linear SVC would perform quickly and sufficiently on the data sets. Thus, the train data was used to train a SVM using SciPy Linear SVC with default settings. The trained model and scaler were saved. The model was then tested using the test set. The test data was scaled using the saved scaler. Metrics were exported including a confusion matrix, classification report, accuracy, F-measure, precision, recall, and time.

The feature selection model, Model B, was then created. The following was executed for each experiment. The settings for the genetic algorithm were set and the termination conditions calculated, based upon the number of generations being iterated through. The initial population was randomly generated. The population was evolved by evaluating the fitness of the population, the selected feature subset's classification accuracy. The same train and test data that was used in Model A was used in Model B. Again, SciPy's Robust Scaler was used as well as SciPy's Linear SVC, both with default settings. The top performers were retained based upon the specified settings; mutations, lower performer inclusions, and cross-overs were performed. The iterations of the genetic algorithm were

continued until termination conditions were met. Exported metrics included for the top performing feature set: the features selected, feature subset model's accuracy, confusion matrix, classification report, accuracy, F-measure, precision, recall, and time. Also exported was the average accuracy for the top ten performing feature subset's for each GA.

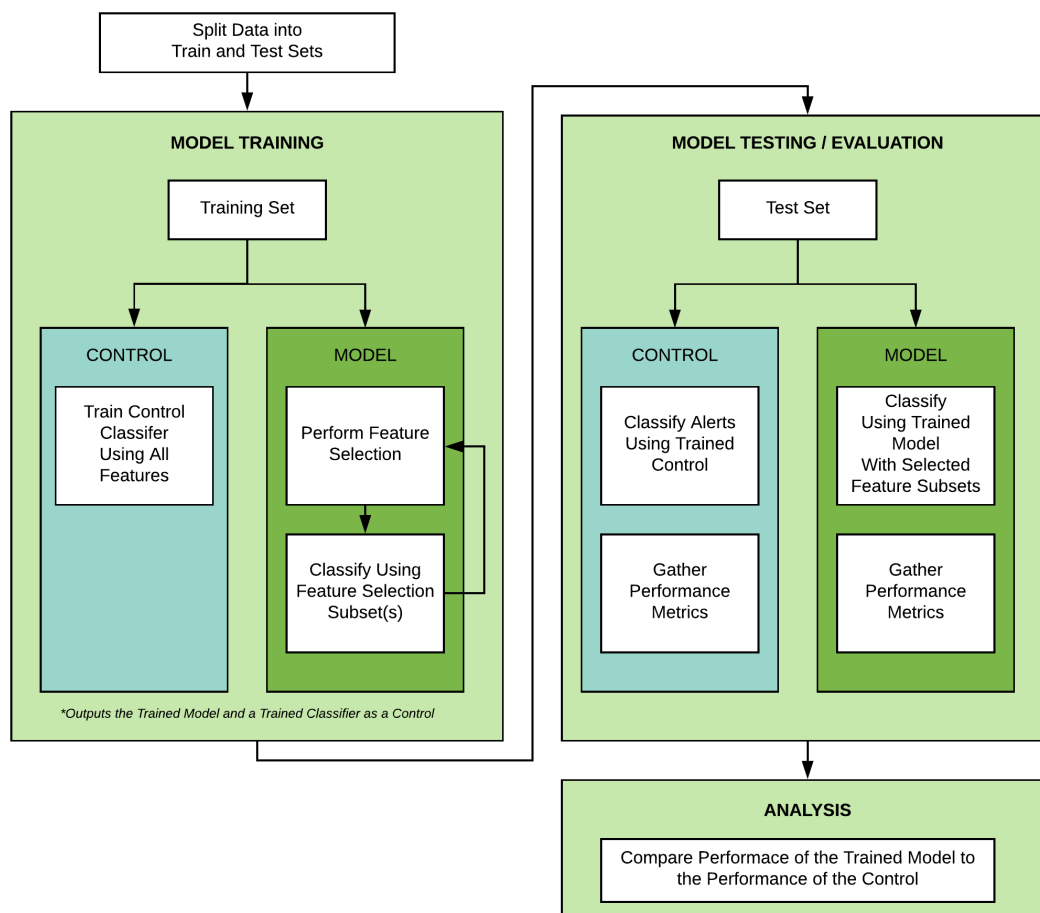
### *Model Validation*

Several experiments were designed and executed to validate the feature selection model's performance. These included a test for each project, a test for all projects together, a test for version predictions, and a test for cross-project predictions. The model's performance metrics for each experiment was compared with the control classifier's performance on all features.

Table 7 outlines the experiments that were performed. Figure 5 shows the experiment process. Features excluded during pre-processing for each experiment are listed in Appendix D. Results for all experiments are presented in Chapter 4.

*Table 7* Experiment, Project, and Data Set Used

Experiment	Project	Data Set
Experiment 1	Models Performance on PhpMyAdmin	PhpMyAdmin
Experiment 2	Models Performance on Moodle	Moodle
Experiment 3	Models Performance on Drupal	Drupal
Experiment 4	Models Performance on Cross Project	Train: PhpMyAdmin and Moodle Test: Drupal
Experiment 5	Models Performance on Version Prediction	Train: PhpMyAdmin Alert Data for v2.2.0 to v3.4.9 Test: PhpMyAdmin Alert Data for v3.5.0 to 4.0.9
Experiment 6	Models Performance on Combined Projects	All



*Figure 5 Model Training, Testing, and Analysis*

## Resources

This research effort required software, hardware, and data sets.

The software programs used for this research included SonarQube, Python, R, Understand, and two commercial static analysis tools. SonarQube is an open source static code analysis application. Python is an open source programming language capable of running on most operating systems. There are machine learning libraries that can perform classification, regression, and clustering (Python Software Foundation, n.d.). R is a popular program in the research community and has been used in several published

works (R Core Team, 2013). Understand is a software metrics and analysis tool (Understand, n.d.).

For hardware, sufficient memory, processors, and storage to run SA tools and data analysis models was necessary. Each SA application had specific hardware requirements minimums. Servers that had commercial SA tools installed exceeding the commercial tools hardware requirements were used for the scanning of the code bases using the commercial tools. A stand-alone MacOSX quad core with 2.8 GHz processors with 16GB of RAM, with over 1 TB of storage was also used for this research effort. This machine was used for SonarQube scans, all Python scripting, and for all model building, testing, and evaluation.

All data sets that were evaluated were downloaded as well as the original version of the PHP Security Vulnerability Dataset. Data sets used for the model's evaluation were generated using the methods previously outlined.

## **Summary**

A feature selection method to assist in the classification of static analysis alerts was presented. Candidate test suites and data sets were evaluated. An existing data set was selected and enhanced. A detailed and literature justified framework for both the data sets usage and enhancement method was thoroughly investigated, described, and executed. Several experiments were outlined and performed to evaluate the feature selection model's performance.

## **Chapter 4**

### **Results**

All experiments outlined in Chapter 3 were performed and the results exported. The control classifier is referred to as Model A. The feature selection model is referred to as Model B. A full list of both model's metrics for each experiment is listed in Appendix B and Appendix C as well as a full list of the Top Performing Feature Subsets in Appendix E. The results are presented herein.

During initial testing, some models were learning on particular features erroneously. Specifically, version last seen and alert lifetime were sensitive features for some models. This was because in those data sets, the values for unactionable alerts had similar values while actionable alerts had distinct values. All features dropped for each experiment were logged and are presented in Appendix D.

#### **Experiment 1**

This experiment tested the model's performance on static analysis alerts from 95 versions of PhpMyAdmin from version 2.2.0 to version 4.0.9. This data set included 12,296 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 84.52% and a false positive rate of 11.85% on 172 features while Model B's best performance had an accuracy of 89.9% and a false positive rate of 10.66% from utilizing only 94 features.

There was an increased accuracy of 6.35%, a decrease in the false positive rate of 10%, and a feature set reduction of 45.35%.



*Table 8 Experiment 1 Results*

	Model A	Model B	% Change
Accuracy	84.52	89.90	6.35
Recall	84.52	89.90	6.35
Precision	84.85	89.90	5.95
F-Measure	84.49	89.90	6.40
FPR	11.85	10.66	-10.04
Feature Count	172	94	- 45.35

*\*numbers in percentages*

## Experiment 2

This experiment tested the model's performance on static analysis alerts from 71 versions of Moodle from version 1.0.0 to version 2.6.1. This data set included 23,398 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 65.72% and a false positive rate of 38.6% on 172 features while Model B's best performance had an accuracy of 83.24% and a false positive rate of 21.61% from utilizing only 80 features.

There was an increased accuracy of 26.66%, a decrease in the FPR of 44%, and a feature set reduction of 53.49%.

*Table 9 Experiment 2 Results*

	Model A	Model B	% Change
Accuracy	65.72	83.24	26.66
Recall	65.91	83.24	26.29
Precision	67.84	84.17	24.07
F-Measure	64.84	83.15	28.24
FPR	38.60	21.61	-44.02
Feature Count	172	80	-53.49

*\*numbers in percentages*

### Experiment 3

Experiment 3 tested the model's performance on static analysis alerts from 38 versions of Drupal from version 6.0.0 to version 6.38. This data set included 960 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 69.72% and a false positive rate of 36.28% on 162 features while Model B's best performance had an accuracy of 83.6% and a false positive rate of 20.93% from utilizing only 85 features.

There was an increased accuracy of 19.91%, a decrease in the FPR of 42.31%, and a feature set reduction of 47.53%.

*Table 10 Experiment 3 Results*

	Model A	Model B	% Change
Accuracy	69.72	83.60	19.91
Recall	70.34	83.60	18.85
Precision	72.13	84.22	16.76
F-Measure	69.25	83.57	20.68
FPR	36.28	20.93	-42.31
Feature Count	162	85	-47.53

*\*numbers in percentages*

### Experiment 4

Experiment 4 tested the model's performance on static analysis alerts across projects. Models were trained using PhpMyAdmin and Moodle data and then tested on Drupal data. This data set included 21,186 alerts. This experiment was the longest running experiment. Improvement was not anticipated for this test. There was an improvement in the classification accuracy of alerts using the feature selection model; however, there was also an increase in the false positive rate.

Model A had an accuracy of 63.54% and a false positive rate of 16.5% on 174 features while Model B's best performance had an accuracy of 70.63% and a false positive rate of 22.5% from utilizing only 79 features.

There was an increased accuracy of 11.16%, an increase in the FPR of 36.36%, and a feature set reduction of 54.6%.

This was the only experiment that increased the false positive rate. The increased accuracy was not consistent amongst all GAs tested for this experiment. Full results for each GA tested is in Appendix C.

*Table 11 Experiment 4 Results*

	Model A	Model B	% Change
Accuracy	63.54	70.63	11.16
Recall	63.54	70.63	11.16
Precision	71.00	72.00	1.41
F-Measure	59.99	70.16	16.95
FPR	16.50	22.50	36.36
Feature Count	174	79	-54.60

*\*numbers in percentages*

## Experiment 5

This experiment tested the model's performance on static analysis alerts for version prediction of PhpMyAdmin. Models were trained on data from PhpMyAdmin versions 2.2.0 to 3.4.9 and then tested on data from PhpMyAdmin versions 3.5.0 to 4.0.9. This data set included 32,499 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 69.49% and a false positive rate of 20% on 174 features while Model B's best performance had an accuracy of 82.92% and a false positive rate of 16.71% from utilizing only 75 features.

There was an increased accuracy of 19.33%, a decrease in the FPR of 16.49%, and a feature set reduction of 56.9%.

*Table 12 Experiment 5 Results*

	Model A	Model B	% Change
Accuracy	69.49	82.92	19.33
Recall	69.49	82.92	19.33
Precision	70.99	82.78	16.61
F-Measure	70.04	82.12	17.25
FPR	20.01	16.71	-16.49
Feature Count	174	75	-56.90

*\*numbers in percentages*

## Experiment 6

This experiment tested the model's performance on static analysis alerts across all three projects. Due to the imbalance in alert quantities from the project specific data sets, two tests were performed. One test was performed on a data set that randomly selected alerts from the main project data sets disregarding imbalanced alert quantities, 6A. Another test was performed on a data set that ensured that there were equal alerts from each project represented in the data set, 6B.

For 6A, the data set included 20,390 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 74.94% and a false positive rate of 30.67% on 174 features while Model B's best performance had an accuracy of 82.52% and a false positive rate of 20.18% from utilizing only 65 features.

There was an increased accuracy of 10.12%, a decrease in the FPR of 34.2%, and a feature set reduction of 62.64%.

*Table 13 Experiment 6A Results*

	Model A	Model B	% Change
Accuracy	74.94	82.52	10.12
Recall	75.00	82.52	10.03
Precision	77.14	82.78	7.31
F-Measure	74.45	82.49	10.80
FPR	30.67	20.18	-34.20
Feature Count	174	65	-62.64

*\*numbers in percentages*

For 6B, the data set included 2,880 alerts. There was an improvement in the classification accuracy of alerts using the feature selection model. Model A had an accuracy of 77.71% and a false positive rate of 21.43% on 174 features while Model B's best performance had an accuracy of 81.49% and a false positive rate of 18.75% from utilizing only 87 features.

There was an increased accuracy of 4.86%, a decrease in the FPR of 12.51%, and a feature set reduction of 50%.

*Table 14 Experiment 6B Results*

	Model A	Model B	% Change
Accuracy	77.71	81.49	4.86
Recall	77.71	81.49	4.86
Precision	77.73	81.50	4.85
F-Measure	77.70	81.49	4.88
FPR	21.43	18.75	-12.51
Feature Count	174	87	-50.00

*\*all numbers in percentages*

## Research Questions Answered

The research questions were answered based on the results from the experiments. This included the model performance and feature subset commonalities. Below are the

answers to the research questions asked. For reference, a list of the Top Performing Feature Subsets for all experiments are presented in Appendix E.

1. Did the proposed model improve the classification of alerts?

Yes. The lowest performing experiment, Experiment 6B, resulted in an accuracy improvement of 4.86% and a false positive rate improvement of 12.51%. This was accomplished on a feature set reduced by 50%. The best performing experiment, Experiment 2, resulted in an accuracy improvement of 26.66% and a false positive rate improvement of 44%. This was accomplished on a feature set reduced by 53.49%.

2. Did selected feature subsets from the proposed model vary between projects?

Yes, feature subsets did vary between projects.

3. Were some features never selected?

Only one feature was not selected by any top performing feature subset, destination file to version alert line of code ratio.

4. Similarly, were some features always selected?

Yes. In all experiments the top performing feature subset included the matched CVE Boolean. A complete list of top performing feature subsets is presented in Appendix E.

## Chapter 5

### Conclusions and Summary

#### Conclusions

This research evidenced that the feature selection methodologies do increase the classification accuracy and reduce the false positive rate in the classification of static analysis alerts. In particular, genetic feature selection methodologies showed statistically significant increases in the classification accuracy of alerts over a model leveraging all features. In brief, the feature selection model presented showed increase accuracy on the classification of alerts on a reduced feature set.

The only feature that was selected in all experiments was the CVE matched Boolean. The second highest selected features in 6 of the 7 experiments were: the destination file's line, and sum Cyclomatic modified; in OWASP 2013 Boolean; and the source file's average blank lines, average lines of code, line count, count of declarative statements, and JavaScript count of declarative statements.

There was only one feature that was never selected outside of dropped features and that was the destination file to version alert line of code ratio. Other less popular features only selected in one experiment were: the source and destination file deleted lines; the destination file original data sets Cyclomatic complexity, average cyclomatic strict, count of declarative files, path count, ratio of comment to code; the OWASP 2017 Boolean; and the source file Halstead's volume, percent modified, average Cyclomatic modified, count of blank HTML lines, code line count, path count, count of executable PHP statements, and maximum nesting complexity.

The feature selection model performed well on the three experiments that tested same project alert classification. Experiments 1, 2, and 3 showed accuracy improvements of 6%, 27%, and 20% with false positive rate reductions of 10%, 44% and 42%. The reduction in feature sets was also significant. The features sets were reduced by 45%, 53%, and 48%, respectively. This validates that the feature selection model improved the classification accuracy and reduced the false positive rate by training models using around half of the features from the original set specified.

The feature selection model also performed well for version prediction. Experiment 5 had an accuracy improvement of 19.33% and a false positive rate reduction of 16.49%. This was on a feature set that was reduced by 56.9%. This indeed showed that new version static analysis alerts may be classified using previous version alerts with a significantly reduced feature set.

Interestingly, the feature selection model increased accuracy and but also increased the false positive rate. On cross-project prediction, the model showed an increase in accuracy over 11% on a feature set reduced by 55%.

It was shown that the feature selection model also improved classification accuracy on data sets that combined alerts from several projects. The accuracy increased by 10% and the false positive rate was reduced by 34% using only 62% of the features.

For each experiment the model was tested on several genetic algorithm instances utilizing different settings for population size, generations, cross-over rates, mutation rates, and termination conditions. Of the multiple tests per experiment, there were only a few instances in which the model did not outperform the control. These were in Experiments 4 and 6B. This is interesting because Experiment 4 was the cross-project



test. Improvement was not anticipated; however, 40% of the genetic tests performed showed better accuracy while using less than half of the features. In Experiment 6B, only two of the 12 genetic tests performed did not show improvement in accuracy over the control; however, the accuracies of those two instances were within one percentage point of the control yet used around half of the features. In all other experiments, the feature selection model always outperformed the control.

Output from the experiments included the average accuracy of the top ten performing feature sets, chromosomes, for each test. If the average of the top ten chromosomes was significantly different than the top performing chromosome, it could be determined that the feature set is unique. Similarly, if the average accuracies and the top performer accuracies are closer in range, it may indicate that there are several feature sets that would produce similar accuracies. Results indicate that the average accuracies of the top ten chromosomes in the final populations were similar to the top performing chromosomes, or feature sets, in the final populations. This indicates that there are multiple feature sets that may provide similar accuracy results. The Average Top Ten Accuracy metric may be found in Appendix C.

In summary, the feature selection model performed well on all tests performed. By using this model to classify alerts generated from static analysis tools, there can be increased confidence in their classifications. Additionally, more focus could be placed on gathering relevant predictive features rather than irrelevant features. In practice, by gathering these additional predictive features and implementing this feature selection model, static analysis alerts could be more confidently classified prior to the analysis of scan results or any alert investigation is commenced.

This study is presented with limitations. A main limitation was the creation of the data set. Although an initial data set was used in previous research, the addition of the static analysis alerts changed the composition, features, and size of the data set significantly. Automatic labeling of alerts was performed based on historical data and some manual investigation performed. Ideally, every alert in the data set should be manually verified as true or false. Thus, although this work provides a data set, future research into creating a labeled real-world static analysis alert data set should ensue. Perhaps the data set could be improved as a future update. However, this data set offers a starting point in the pursuit of a real-world labeled static analysis data set. It allows future research to utilize this data set to compare static analysis machine learning models to one another.

Another limitation was the analysis of projects containing mainly one programming language. Ancillary languages were included for instance JavaScript and HTML; however, the main language for the analysis was PHP.

### **Implications and Recommendations**

The impacts of this work on the static analysis domain is meaningful. This work has presented a real-world static analysis data set based upon three open source PHP applications that may be used in future research efforts. As part of this work, this data set is now published and freely available for other researchers use. This is a significant addition to the domain. Additionally, the framework for generating static analysis data sets has been enhanced to include additional security related features. This framework could also be used to create other real-world static analysis data sets. Continued research that produces and publishes labeled real-world static analysis data sets is still needed.

A significant addition to the knowledge base is the evidence that software metric features were consistently selected as relevant features in the improved classification accuracy of static analysis alerts. The static analysis literature has often focused on histories with limited file or software metrics. Further research to explore the association between software code metrics and static analysis alert classification could prove promising.

The main contribution of this work, however, is a feature selection method that improves the classification of static analysis alerts, ergo, reducing the false positive rate. Research of other feature selection methods and the further investigation into relevant feature sets for static analysis alert classification could be further explored.

## **Summary**

It is imperative that software being developed is secure and free from security vulnerabilities and bugs. One method to assist in detecting insecure code is to perform static code analysis. Currently, static analysis tools present developers with a high amount of false positive and unactionable alerts. The goal of this research effort was to develop and evaluate methods for feature selections that helped to improve the classification accuracy of static analysis alerts; thereby, addressing the problem of high false positive rates. This research effort presented and tested a novel method leveraging feature selection that resulted in the improved classification of alerts.

A review of the extant literature and history of the static analysis domain was performed. The domain's current problems were discussed. Motivating factors for the continued research were clearly outlined and a goal for the research effort was posited.

Research questions that guided this research were presented and answered. Barriers, limitations, and assumptions were identified.

A genetic feature selection method was presented as a potential solution. Genetic algorithm methodologies were explored and the model's specifications outlined. A support vector machine classification method was chosen and a review of the classifier specifics was presented.

Candidate test suites and data sets were evaluated. An existing data set was selected and enhanced. A detailed and literature justified framework for both the data sets usage and enhancement method was thoroughly investigated, described, and executed. The data set generation process was followed with some additions. In addition, the CVEs, release notes, security notices, and change logs were gathered, and the bug report systems queried. This additional information was used to assist in alert labeling and feature generation.

From the original dataset, Drupal versions were expanded to include 38 versions from 6.0.0 to 6.38. The PhpMyAdmin and Moodle versions evaluated remained unchanged. 397 additional CVEs were discovered.

Each release of each application was scanned by each tool and the resulting alerts labeled. Work was performed to match auxiliary information such as change logs, security notices, release notes, bug tracking systems, and CVEs to the alerts. For labeling, alerts were tracked between versions and auxiliary information inspected. Additional software metrics were gathered and merged to augment the data set to create a static analysis data set.

The result of this process was a data set for each project containing labeled real-world static analysis alerts from three static analysis scanners, complete with source code metrics and historical features of the alerts. The raw data sets totaled 256,198 alerts.

Prior to running experiments, the respective raw data sets were pre-processed. After pre-processing the data sets were split into train and test sets. Once train and test sets were created, the experiments were performed.

Experiments were executed to train and evaluate the feature selection model. The model's selected feature subsets and classification performance metrics were exported for each experiment. To measure improved classification accuracy, the feature selection model was compared with a similar model that excluded the feature selection component. In essence, for each experiment, a SVM classifier was trained and tested on the same train and test data sets and compared to the feature selection model's performance.

Several experiments ensued. These included a test for each project, a test for all projects in aggregate, a version prediction test, and a cross-project predication test. The model's performance metrics for each experiment was compared with the control classifier's performance on all features.

Results were presented showing increased classification accuracies and lower false positive rates in all experiments using reduced feature sets generated using a genetic algorithm. It was shown that predictions could be made about alert classifications within the same projects. It was also shown that alert classification predictions could be made on future project versions. Interestingly, the model even showed improvements on cross-project alert classification predictions; however, the projects tested were all of a similar language and structure as they were all open source PHP applications.

Conclusions, limitations, implications, and impacts to the domain were discussed. Some directions of future research were identified.

Succinctly, a feature selection method was presented, developed, and evaluated. A data set was selected and enhanced. The final data set was composed of static analysis alerts generated from three scanning tools on the source code of three open source PHP projects spanning several years. Once data was gathered and preprocessed, the data was split into train and test sets. A genetic feature selection model was trained and tested on the train and test sets respectively. The process was performed iteratively, testing selected feature subsets for an improvement in classification accuracy in an embedded fashion. This process resulted in a subset of relevant features for the classification of the alerts. The results were compared to a classification model, sans feature selection, to quantify the classification improvement of the feature selection model. There were statistically significant improvements in the classification accuracy of the alerts using a reduced feature set. Therefore, feature selection methods can be used to increase the classification accuracy of static analysis alerts and, thereby reduce the false positive rate.

## Appendix A

### List and Descriptions of Features

FEATURE NAME	ORIGIN	DESCRIPTION
ALERT_LIFETIME	Engineered	Age of the alert.
CATEGORY	Alert	Category of the warning.
CODE_BUG_VULN	Alert	Type of alert.
COMMITTED_DATE	Git	The commit date of the file.
CVE_ID	Engineered	The CVE id associated with the alert.
CWE_ID	Alert	The CWE associated with the alert.
DEST_CREATED_DATE	Git	The date the file was created.
DEST_FILE_AGE	Engineered	Age of file from creation date.
DEST_FILE_CHURN	Engineered	Sum of lines added, modified, and deleted.
DEST_FILE_CLOC	Engineered	Number of commented lines of code.
DEST_FILE_COLUMN	Alert	Column of the variable, the location on the line.
DEST_FILE_COMPLETE	Alert	The complete file name including the path.
DEST_FILE_EDIT_FREQUENCY	Engineered	Number of times a file has been edited.
DEST_FILE_ELOC	Engineered	Empty lines of code.
DEST_FILE_EXT	Alert	Extension / type of the file.
DEST_FILE_FOLDER	Alert	Immediate folder the file lives in.
DEST_FILE_FUNCTION_VAR	Alert	Function or method the alert is originating from.
DEST_FILE_GROWTH	Engineered	Difference between lines added and deleted.
DEST_FILE_LINE	Alert	Destination line of code
DEST_FILE_LINES_ADDED	Git	Number of lines added.
DEST_FILE_LINES_DELETED	Git	Number of lines deleted.
DEST_FILE_LOC	Engineered	Lines of code in the file.
DEST_FILE_NAME	Alert	Name of the file.
DEST_FILE_orig_ccom	Dataset	Cyclomatic complexity, the number of independent paths through a function.
DEST_FILE_orig_ccomdeep	Dataset	Deep cyclomatic complexity.
DEST_FILE_orig_hvol	Dataset	Halstead's Volume estimate $((N1 + N2) \log n1 + n1)$ using the number of unique operators (n1) and operands (n1) and the number of total operators (N1) and operands (N2) in the file.
DEST_FILE_orig_loc	Dataset	Lines of code in the file.
DEST_FILE_orig_nest	Dataset	Maximum depth for nested loops and control structures in the file.
DEST_FILE_orig_nIncomingCalls	Dataset	Number of incoming calls.
DEST_FILE_orig_nIncomingCallsUniq	Dataset	Number of unique incoming calls.
DEST_FILE_orig_nmethods	Dataset	Number of methods.
DEST_FILE_orig_nonecholoc	Dataset	Number of empty lines of code.
DEST_FILE_orig_nOutgoingExternCallsUniq	Dataset	Number of unique external calls.
DEST_FILE_orig_nOutgoingExternFlsCalled	Dataset	Number of external files called.
DEST_FILE_orig_nOutgoingExternFlsCalledUniq	Dataset	Number of unique external files called.
DEST_FILE_orig_nOutgoingInternCalls	Dataset	Number of internal calls.
DEST_FILE_PATH	Alert	Complete directory path of the file.
DEST_FILE_PERCENT_MODIFIED	Engineered	Percent of total modified lines.
DEST_FILE_SIZE	Engineered	Size of the file.
DEST_FILE_STALENESS	Engineered	Time from last change of the file.
DEST_FILE_VERSION_ALERT_COUNT	Engineered	Number of alerts for the file.
DEST_FILE_VERSION_ALERT_LOC_RATIO	Engineered	Percent of alerts in the file to the LOC of the file.
DEST_LAST_AUTHOR_EMAIL	Git	The files last authors email.
DEST_LAST_AUTHOR_NAME	Git	The files last authors name.
DEST_LAST_EDITED_DATE	Git	The files last edited date.

DEST_UNDRSTD_AvgCyclomatic	Metrics Tool	Average cyclomatic complexity for all nested functions or methods.
DEST_UNDRSTD_AvgCyclomaticModified	Metrics Tool	Average modified cyclomatic complexity for all nested functions or methods.
DEST_UNDRSTD_AvgCyclomaticStrict	Metrics Tool	Average strict cyclomatic complexity for all nested functions or methods.
DEST_UNDRSTD_AvgEssential	Metrics Tool	Average strict cyclomatic complexity for all nested functions or methods.
DEST_UNDRSTD_AvgLine	Metrics Tool	Average number of lines for all nested functions or methods.
DEST_UNDRSTD_AvgLineBlank	Metrics Tool	Average number of blank for all nested functions or methods.
DEST_UNDRSTD_AvgLineCode	Metrics Tool	Average number of lines containing source code for all nested functions or methods.
DEST_UNDRSTD_AvgLineComment	Metrics Tool	Average number of lines containing comment for all nested functions or methods.
DEST_UNDRSTD_CountDeclClass	Metrics Tool	Number of classes.
DEST_UNDRSTD_CountDeclExecutableUnit	Metrics Tool	Executable Statements
DEST_UNDRSTD_CountDeclFile	Metrics Tool	Number of files.
DEST_UNDRSTD_CountDeclFunction	Metrics Tool	Number of functions.
DEST_UNDRSTD_CountLine	Metrics Tool	Number of all lines.
DEST_UNDRSTD_CountLine_Html	Metrics Tool	Number of all html lines.
DEST_UNDRSTD_CountLine_Javascript	Metrics Tool	Number of all javascript lines.
DEST_UNDRSTD_CountLine_Php	Metrics Tool	Number of all php lines.
DEST_UNDRSTD_CountLineBlank	Metrics Tool	Number of blank lines.
DEST_UNDRSTD_CountLineBlank_Html	Metrics Tool	Number of blank html lines.
DEST_UNDRSTD_CountLineBlank_Javascript	Metrics Tool	Number of blank javascript lines.
DEST_UNDRSTD_CountLineBlank_Php	Metrics Tool	Number of blank php lines.
DEST_UNDRSTD_CountLineCode	Metrics Tool	Number of lines containing source code.
DEST_UNDRSTD_CountLineCode_Javascript	Metrics Tool	Number of javascript lines containing source code.
DEST_UNDRSTD_CountLineCode_Php	Metrics Tool	Number of php lines containing source code.
DEST_UNDRSTD_CountLineComment	Metrics Tool	Number of lines containing comment.
DEST_UNDRSTD_CountLineComment_Html	Metrics Tool	Number of html lines containing comment.
DEST_UNDRSTD_CountLineComment_Javascript	Metrics Tool	Number of javascript lines containing comment.
DEST_UNDRSTD_CountLineComment_Php	Metrics Tool	Number of php lines containing comment.
DEST_UNDRSTD_CountPath	Metrics Tool	Number of possible paths, not counting abnormal exits or gotos.
DEST_UNDRSTD_CountPathLog	Metrics Tool	Log10, truncated to an integer value, of the metric CountPath
DEST_UNDRSTD_CountStmt	Metrics Tool	Number of statements.
DEST_UNDRSTD_CountStmtDecl	Metrics Tool	Number of declarative statements.
DEST_UNDRSTD_CountStmtDecl_Javascript	Metrics Tool	Number of javascript declarative statements.
DEST_UNDRSTD_CountStmtDecl_Php	Metrics Tool	Number of php declarative statements.
DEST_UNDRSTD_CountStmtExe	Metrics Tool	Number of executable statements.
DEST_UNDRSTD_CountStmtExe_Javascript	Metrics Tool	Number of javascript executable statements.
DEST_UNDRSTD_CountStmtExe_Php	Metrics Tool	Number of php executable statements.
DEST_UNDRSTD_Cyclomatic	Metrics Tool	Cyclomatic complexity.
DEST_UNDRSTD_CyclomaticModified	Metrics Tool	Modified cyclomatic complexity.
DEST_UNDRSTD_CyclomaticStrict	Metrics Tool	Strict cyclomatic complexity.
DEST_UNDRSTD_Essential	Metrics Tool	Essential complexity.
DEST_UNDRSTD_MaxCyclomatic	Metrics Tool	Maximum cyclomatic complexity of all nested functions or methods.
DEST_UNDRSTD_MaxCyclomaticModified	Metrics Tool	Maximum modified cyclomatic complexity of nested functions or methods.
DEST_UNDRSTD_MaxEssential	Metrics Tool	Maximum essential complexity of all nested functions or methods.
DEST_UNDRSTD_MaxInheritanceTree	Metrics Tool	Maximum depth of class in inheritance tree.
DEST_UNDRSTD_MaxNesting	Metrics Tool	Maximum nesting level of control constructs.
DEST_UNDRSTD_RatioCommentToCode	Metrics Tool	Ratio of comment lines to code lines.
DEST_UNDRSTD_SumCyclomatic	Metrics Tool	Sum of cyclomatic complexity of all nested functions or methods.
DEST_UNDRSTD_SumCyclomaticModified	Metrics Tool	Sum of modified complexity of all nested functions or methods.
DEST_UNDRSTD_SumCyclomaticStrict	Metrics Tool	Sum of strict cyclomatic complexity of all nested functions or methods.



DEST_UNDRSTD_SumEssential	Metrics Tool	Sum of essential complexity of all nested functions or methods.
FILES_IN_PATH	Engineered	Number of files the path of the alert traverses through.
IN_OWASP_2013	Alert	Is alert in the top ten OWASP 2013.
IN_OWASP_2017	Alert	Is alert in the top ten OWASP 2017.
LANGUAGE	Alert	The programming language generating the alert.
MATCHED_CVE	Engineered	Is there an associated CVE with the alert.
NUM_PATH_HOPS	Engineered	Then number of hops in the alert path from source to destination.
OWASP_TOP_TEN_2013	Alert	The OWASP Top Ten 2013 category.
OWASP_TOP_TEN_2017	Alert	The OWASP Top Ten 2017 category.
PRIORITY	Alert	Priority of the alert from tool.
PROJECT	Scan	Project the alert resides in.
SOURCE_CREATED_DATE	Git	The date the file was created.
SOURCE_DEST_SAME_FILE	Engineered	Are the source and destination the same.
SOURCE_FILE_AGE	Engineered	Age of file from creation date.
SOURCE_FILE_CHURN	Engineered	Sum of lines added, modified, and deleted.
SOURCE_FILE_CLOC	Engineered	Number of commented lines of code.
SOURCE_FILE_COLUMN	Alert	Column of the variable, the location on the line.
SOURCE_FILE_COMPLETE	Alert	The complete file name including the path.
SOURCE_FILE_EDIT_FREQUENCY	Engineered	Number of times a file has been edited.
SOURCE_FILE_ELOC	Engineered	Empty lines of code.
SOURCE_FILE_EXT	Alert	Extension / type of the file.
SOURCE_FILE_FOLDER	Alert	Immediate folder the file lives in.
SOURCE_FILE_FUNCTION_VAR	Alert	Function or method the alert is originating from.
SOURCE_FILE_GROWTH	Engineered	Difference between lines added and deleted.
SOURCE_FILE_LINE	Alert	Source line of code.
SOURCE_FILE_LINES_ADDED	Git	Number of lines added.
SOURCE_FILE_LINES_DELETED	Git	Number of lines deleted.
SOURCE_FILE_LOC	Engineered	Lines of code in the file.
SOURCE_FILE_NAME	Alert	Name of the file.
SOURCE_FILE_orig_ccom	Dataset	Cyclomatic complexity, the number of independent paths through a function.
SOURCE_FILE_orig_ccomdeep	Dataset	Deep cyclomatic complexity.
SOURCE_FILE_orig_hvol	Dataset	Halstead's Volume estimate $((N1 + N2) \log n1 + n1)$ using the number of unique operators (n1) and operands (n1) and the number of total operators (N1) and operands (N2) in the file.
SOURCE_FILE_orig_loc	Dataset	Lines of code in the file.
SOURCE_FILE_orig_nest	Dataset	Maximum depth for nested loops and control structures in the file.
SOURCE_FILE_orig_nIncomingCalls	Dataset	Number of incoming calls.
SOURCE_FILE_orig_nIncomingCallsUniq	Dataset	Number of unique incoming calls.
SOURCE_FILE_orig_nmethods	Dataset	Number of methods.
SOURCE_FILE_orig_nonecholoc	Dataset	Number of empty lines of code.
SOURCE_FILE_orig_nOutgoingExternCallsUniq	Dataset	Number of unique external calls.
SOURCE_FILE_orig_nOutgoingExternFlsCalled	Dataset	Number of external files called.
SOURCE_FILE_orig_nOutgoingExternFlsCalledUniq	Dataset	Number of unique external files called.
SOURCE_FILE_orig_nOutgoingInternCalls	Dataset	Number of internal calls.
SOURCE_FILE_PATH	Alert	Complete directory path of the file.
SOURCE_FILE_PERCENT_MODIFIED	Engineered	Percent of total modified lines.
SOURCE_FILE_SIZE	Engineered	Size of the file.
SOURCE_FILE_STALENESS	Engineered	Time from last change of the file.
SOURCE_FILE_VERSION_ALERT_COUNT	Engineered	Number of alerts for the file.
SOURCE_FILE_VERSION_ALERT_LOC_RATIO	Engineered	Percent of alerts in the file to the LOC of the file.
SOURCE_LAST_AUTHOR_EMAIL	Git	The files last authors email.
SOURCE_LAST_AUTHOR_NAME	Git	The files last authors name.
SOURCE_LAST_EDITED_DATE	Git	The files last edited date.
SOURCE_UNDRSTD_AvgCyclomatic	Metrics Tool	Average cyclomatic complexity for all nested functions or methods.

SOURCE_UNDRSTD_AvgCyclomaticModified	Metrics Tool	Average modified cyclomatic complexity for all nested functions or methods.
SOURCE_UNDRSTD_AvgCyclomaticStrict	Metrics Tool	Average strict cyclomatic complexity for all nested functions or methods.
SOURCE_UNDRSTD_AvgEssential	Metrics Tool	Average strict cyclomatic complexity for all nested functions or methods.
SOURCE_UNDRSTD_AvgLine	Metrics Tool	Average number of lines for all nested functions or methods.
SOURCE_UNDRSTD_AvgLineBlank	Metrics Tool	Average number of blank for all nested functions or methods.
SOURCE_UNDRSTD_AvgLineCode	Metrics Tool	Average number of lines containing source code for all nested functions or methods.
SOURCE_UNDRSTD_AvgLineComment	Metrics Tool	Average number of lines containing comment for all nested functions or methods.
SOURCE_UNDRSTD_CountDeclClass	Metrics Tool	Number of classes.
SOURCE_UNDRSTD_CountDeclExecutableUnit	Metrics Tool	Executable Statements
SOURCE_UNDRSTD_CountDeclFile	Metrics Tool	Number of files.
SOURCE_UNDRSTD_CountDeclFunction	Metrics Tool	Number of functions.
SOURCE_UNDRSTD_CountLine	Metrics Tool	Number of all lines.
SOURCE_UNDRSTD_CountLine_Html	Metrics Tool	Number of all html lines.
SOURCE_UNDRSTD_CountLine_Javascript	Metrics Tool	Number of all javascript lines.
SOURCE_UNDRSTD_CountLine_Php	Metrics Tool	Number of all php lines.
SOURCE_UNDRSTD_CountLineBlank	Metrics Tool	Number of blank lines.
SOURCE_UNDRSTD_CountLineBlank_Html	Metrics Tool	Number of blank html lines.
SOURCE_UNDRSTD_CountLineBlank_Javascript	Metrics Tool	Number of blank javascript lines.
SOURCE_UNDRSTD_CountLineBlank_Php	Metrics Tool	Number of blank php lines.
SOURCE_UNDRSTD_CountLineCode	Metrics Tool	Number of lines containing source code.
SOURCE_UNDRSTD_CountLineCode_Javascript	Metrics Tool	Number of javascript lines containing source code.
SOURCE_UNDRSTD_CountLineCode_Php	Metrics Tool	Number of php lines containing source code.
SOURCE_UNDRSTD_CountLineComment	Metrics Tool	Number of lines containing comment.
SOURCE_UNDRSTD_CountLineComment_Html	Metrics Tool	Number of html lines containing comment.
SOURCE_UNDRSTD_CountLineComment_Javascript	Metrics Tool	Number of javascript lines containing comment.
SOURCE_UNDRSTD_CountLineComment_Php	Metrics Tool	Number of php lines containing comment.
SOURCE_UNDRSTD_CountPath	Metrics Tool	Number of possible paths, not counting abnormal exits or gotos.
SOURCE_UNDRSTD_CountPathLog	Metrics Tool	Log10, truncated to an integer value, of the metric CountPath
SOURCE_UNDRSTD_CountStmt	Metrics Tool	Number of statements.
SOURCE_UNDRSTD_CountStmtDecl	Metrics Tool	Number of declarative statements.
SOURCE_UNDRSTD_CountStmtDecl_Javascript	Metrics Tool	Number of javascript declarative statements.
SOURCE_UNDRSTD_CountStmtDecl_Php	Metrics Tool	Number of php declarative statements.
SOURCE_UNDRSTD_CountStmtExe	Metrics Tool	Number of executable statements.
SOURCE_UNDRSTD_CountStmtExe_Javascript	Metrics Tool	Number of javascript executable statements.
SOURCE_UNDRSTD_CountStmtExe_Php	Metrics Tool	Number of php executable statements.
SOURCE_UNDRSTD_Cyclomatic	Metrics Tool	Cyclomatic complexity.
SOURCE_UNDRSTD_CyclomaticModified	Metrics Tool	Modified cyclomatic complexity.
SOURCE_UNDRSTD_CyclomaticStrict	Metrics Tool	Strict cyclomatic complexity.
SOURCE_UNDRSTD_Essential	Metrics Tool	Essential complexity.
SOURCE_UNDRSTD_MaxCyclomatic	Metrics Tool	Maximum cyclomatic complexity of all nested functions or methods.
SOURCE_UNDRSTD_MaxCyclomaticModified	Metrics Tool	Maximum modified cyclomatic complexity of nested functions or methods.
SOURCE_UNDRSTD_MaxEssential	Metrics Tool	Maximum essential complexity of all nested functions or methods.
SOURCE_UNDRSTD_MaxInheritanceTree	Metrics Tool	Maximum depth of class in inheritance tree.
SOURCE_UNDRSTD_MaxNesting	Metrics Tool	Maximum nesting level of control constructs.
SOURCE_UNDRSTD_RatioCommentToCode	Metrics Tool	Ratio of comment lines to code lines.
SOURCE_UNDRSTD_SumCyclomatic	Metrics Tool	Sum of cyclomatic complexity of all nested functions or methods.
SOURCE_UNDRSTD_SumCyclomaticModified	Metrics Tool	Sum of modified complexity of all nested functions or methods.
SOURCE_UNDRSTD_SumCyclomaticStrict	Metrics Tool	Sum of strict cyclomatic complexity of all nested functions or methods.
SOURCE_UNDRSTD_SumEssential	Metrics Tool	Sum of essential complexity of all nested functions or methods.

TOOL	Scan	The tool generating the alert.
TYPE	Alert	The type of alert as specified by the tool.
VERSION_ALERT_COUNT	Scan	The number alerts in the version scan results.
VERSION_LAST_SEEN	Engineered	The last version the alert was seen.
VERSION_START	Scan	The first version the alert was seen.

Appendix B

Experiment Result Metrics

	Model A						Model B Top Performer					
	Accuracy	Recall	Precision	F Measure	FPR	Feature Count	Accuracy	Recall	Precision	F Measure	FPR	Feature Subset Count
Experiment 1	84.52	84.52	84.85	84.49	11.85	172	89.90	89.90	89.90	89.90	10.66	94
Experiment 2	65.72	65.91	67.84	64.84	38.60	172	83.24	83.24	84.17	83.15	21.61	80
Experiment 3	69.72	70.34	72.13	69.25	36.28	162	83.60	83.60	84.22	83.57	20.93	85
Experiment 4*	63.54	63.54	71.00	59.99	16.50	174	70.63	70.63	72.00	70.16	22.50	79
Experiment 5*	69.49	69.49	70.99	70.04	20.01	174	82.92	82.92	82.78	82.12	16.71	75
Experiment 6A	74.94	75.00	77.14	74.45	30.67	174	82.52	82.52	82.78	82.49	20.18	65
Experiment 6B	77.71	77.71	77.73	77.70	21.43	174	81.49	81.49	81.50	81.49	18.75	87

\*weighted

## Appendix C

## GA Performance Metrics

EXP	POPULATION	Gens	Mating	Random Selection	Mutation	Imp. Thres.	Avg. Top 10 Accuracy	Gen. of Best Set	Best Feature Set Accuracy	Number Features	Terminated	Gen Term.
<b>1</b>	<b>200</b>	<b>500</b>	<b>0.7</b>	<b>0.01</b>	<b>0.025</b>	<b>0.003</b>	<b>0.89812716</b>	<b>206</b>	<b>0.898965</b>	<b>94</b>	<b>TRUE</b>	<b>283</b>
1	200	500	0.8	0.05	0.03	0.0003	0.89645145	149	0.89674717	92	TRUE	251
1	150	100	0.75	0.03	0.02	0.003	0.89497289	49	0.89576146	86	TRUE	66
1	200	500	0.75	0.03	0.02	0.003	0.89517003	130	0.89576146	91	TRUE	207
1	150	50	0.8	0.05	0.03	0.0003	0.89231148	36	0.89354362	90	TRUE	48
1	150	50	0.75	0.05	0.03	0.0003	0.89078364	11	0.89280434	93	TRUE	20
1	100	50	0.8	0.05	0.03	0.0003	0.89083292	19	0.89280434	90	TRUE	31
1	100	50	0.7	0.01	0.025	0.003	0.89110399	20	0.89255791	92	TRUE	29
1	150	100	0.7	0.03	0.02	0.003	0.89112863	18	0.8915722	103	TRUE	35
1	50	20	0.7	0.01	0.025	0.003	0.88767866	12	0.8905865	84	TRUE	17
1	50	20	0.75	0.03	0.02	0.003	0.88851651	10	0.8905865	86	TRUE	15
1	100	50	0.75	0.03	0.02	0.003	0.88861508	11	0.8905865	86	TRUE	20
1	150	50	0.7	0.05	0.03	0.0003	0.88945293	8	0.89034007	81	TRUE	17
1	50	20	0.8	0.05	0.03	0.0003	0.88546082	6	0.88639724	85	TRUE	12
<b>2</b>	<b>200</b>	<b>500</b>	<b>0.8</b>	<b>0.05</b>	<b>0.03</b>	<b>0.0003</b>	<b>0.83161098</b>	<b>434</b>	<b>0.83281533</b>	<b>80</b>	<b>TRUE</b>	<b>485</b>
2	150	100	0.75	0.03	0.02	0.003	0.82645688	90	0.82750583	82	FALSE	0
2	150	100	0.7	0.03	0.02	0.003	0.82602953	70	0.82659933	78	TRUE	87
2	100	50	0.75	0.03	0.02	0.003	0.82341362	34	0.82530433	83	TRUE	43
2	150	50	0.8	0.05	0.03	0.0003	0.81801347	28	0.82012432	92	TRUE	40
2	100	50	0.7	0.01	0.025	0.003	0.81253561	9	0.81377881	85	TRUE	18

2	100	50	0.8	0.05	0.03	0.0003	0.81003626	2	0.81339031	78	TRUE	14
2	50	20	0.75	0.03	0.02	0.003	0.80934991	15	0.81248381	88	TRUE	19
2	50	20	0.8	0.05	0.03	0.0003	0.7997928	0	0.80769231	84	TRUE	6
2	50	20	0.7	0.01	0.025	0.003	0.79853665	0	0.80549081	90	TRUE	5
<b>3</b>	<b>200</b>	<b>500</b>	<b>0.8</b>	<b>0.05</b>	<b>0.03</b>	<b>0.0003</b>	<b>0.83312303</b>	<b>54</b>	<b>0.83596215</b>	<b>85</b>	<b>TRUE</b>	<b>156</b>
3	150	200	0.7	0.01	0.025	0.003	0.83059937	32	0.83280757	72	TRUE	74
3	200	500	0.7	0.01	0.025	0.003	0.83280757	80	0.83280757	80	TRUE	182
3	200	500	0.75	0.03	0.02	0.003	0.83280757	78	0.83280757	82	TRUE	180
3	100	50	0.7	0.01	0.025	0.003	0.82429022	31	0.82649842	77	TRUE	43
3	150	100	0.75	0.03	0.02	0.003	0.82113565	34	0.82334385	84	TRUE	46
3	50	20	0.7	0.01	0.025	0.003	0.81104101	4	0.8170347	68	TRUE	10
3	150	50	0.8	0.05	0.03	0.0003	0.81230284	25	0.8170347	90	TRUE	37
3	50	20	0.8	0.05	0.03	0.0003	0.80157729	5	0.81388013	91	TRUE	11
3	100	50	0.75	0.03	0.02	0.003	0.80694006	18	0.81072555	90	TRUE	30
3	100	50	0.8	0.05	0.03	0.0003	0.80883281	17	0.81072555	87	TRUE	29
3	50	20	0.75	0.03	0.02	0.003	0.78990536	1	0.79495268	68	TRUE	7
<b>4</b>	<b>200</b>	<b>500</b>	<b>0.8</b>	<b>0.05</b>	<b>0.03</b>	<b>0.0003</b>	<b>0.70625</b>	<b>340</b>	<b>0.70625</b>	<b>79</b>	<b>TRUE</b>	<b>442</b>
4	150	100	0.7	0.03	0.02	0.005	0.6925	73	0.69375	91	TRUE	90
4	150	100	0.75	0.03	0.02	0.005	0.66145833	26	0.66666667	86	TRUE	43
4	100	50	0.8	0.05	0.03	0.0003	0.63739583	31	0.64583333	75	TRUE	41
4	150	50	0.8	0.05	0.03	0.0003	0.61895833	9	0.628125	84	TRUE	21
4	100	50	0.7	0.01	0.025	0.003	0.59958333	0	0.62291667	83	TRUE	9
4	100	50	0.75	0.03	0.02	0.003	0.610625	42	0.621875	81	FALSE	0
4	50	20	0.7	0.01	0.025	0.003	0.60552083	9	0.61458333	84	TRUE	14
4	50	20	0.8	0.05	0.03	0.0003	0.57677083	15	0.58229167	87	FALSE	0
4	50	20	0.75	0.03	0.02	0.003	0.56677083	0	0.571875	70	TRUE	5
<b>5</b>	<b>200</b>	<b>500</b>	<b>0.75</b>	<b>0.03</b>	<b>0.02</b>	<b>0.003</b>	<b>0.829651692</b>	<b>343</b>	<b>0.8291552</b>	<b>75</b>	<b>TRUE</b>	<b>420</b>

5	200	500	0.7	0.01	0.025	0.003	0.802869819	158	0.819765037	78	TRUE	163
5	200	500	0.8	0.05	0.03	0.0003	0.81774634	303	0.8185654	75	TRUE	405
5	150	100	0.75	0.03	0.02	0.003	0.80976669	72	0.81082982	73	TRUE	89
5	100	50	0.8	0.05	0.03	0.0003	0.79044014	47	0.79453131	93	FALSE	0
5	150	100	0.7	0.03	0.02	0.003	0.78939356	40	0.79113924	77	TRUE	57
5	100	50	0.75	0.03	0.02	0.003	0.78895094	25	0.79113924	85	TRUE	34
5	150	50	0.8	0.05	0.03	0.0003	0.78173244	21	0.7895673	71	TRUE	33
5	100	50	0.7	0.01	0.025	0.003	0.77560602	22	0.78418963	88	TRUE	31
5	50	20	0.75	0.03	0.02	0.003	0.77086539	12	0.77732274	93	TRUE	17
5	150	50	0.75	0.05	0.03	0.0003	0.76498718	11	0.77467527	85	TRUE	20
5	150	50	0.7	0.05	0.03	0.0003	0.76208323	5	0.76648465	81	TRUE	14
5	50	20	0.8	0.05	0.03	0.0003	0.73612559	2	0.74195417	93	TRUE	8
5	50	20	0.7	0.01	0.025	0.003	0.72258625	1	0.73827252	77	TRUE	6
<b>6A</b>	<b>200</b>	<b>500</b>	<b>0.75</b>	<b>0.03</b>	<b>0.02</b>	<b>0.003</b>	<b>0.813722693</b>	<b>215</b>	<b>0.8245234</b>	<b>65</b>	<b>TRUE</b>	<b>292</b>
6A	200	500	0.8	0.05	0.03	0.0003	0.82235102	169	0.82315351	79	TRUE	271
6A	200	500	0.7	0.01	0.025	0.003	0.806244613	36	0.820627136	79	TRUE	68
6A	150	100	0.7	0.03	0.02	0.003	0.81821965	36	0.82032992	96	TRUE	53
6A	150	50	0.75	0.05	0.03	0.0003	0.81749145	18	0.81973547	92	TRUE	27
6A	100	50	0.8	0.05	0.03	0.0003	0.81500966	18	0.81899242	93	TRUE	30
6A	150	50	0.8	0.05	0.03	0.0003	0.81536632	18	0.81884381	84	TRUE	30
6A	100	50	0.75	0.03	0.02	0.003	0.81627285	35	0.81839798	83	TRUE	44
6A	150	100	0.75	0.03	0.02	0.003	0.81584188	42	0.81765493	87	TRUE	59
6A	100	50	0.7	0.01	0.025	0.003	0.81361272	20	0.81572299	91	TRUE	29
6A	50	20	0.7	0.01	0.025	0.003	0.81023926	4	0.81497994	98	TRUE	9
6A	50	20	0.8	0.05	0.03	0.0003	0.80105513	3	0.81483133	107	TRUE	9
6A	150	50	0.7	0.05	0.03	0.0003	0.81016496	3	0.81200773	95	TRUE	12
6A	50	20	0.75	0.03	0.02	0.003	0.79527419	2	0.80472581	75	TRUE	7

<b>6B</b>	<b>200</b>	<b>500</b>	<b>0.7</b>	<b>0.01</b>	<b>0.025</b>	<b>0.003</b>	<b>0.80883281</b>	<b>121</b>	<b>0.8149316</b>	<b>87</b>	<b>TRUE</b>	<b>198</b>
6B	150	200	0.7	0.01	0.025	0.003	0.806204	77	0.80757098	73	TRUE	109
6B	200	500	0.75	0.03	0.02	0.003	0.80672976	111	0.80757098	76	TRUE	188
6B	200	500	0.8	0.05	0.03	0.0003	0.80672976	235	0.80757098	82	TRUE	337
6B	150	100	0.75	0.03	0.02	0.003	0.80105152	60	0.8044164	90	TRUE	77
6B	150	50	0.8	0.05	0.03	0.0003	0.79085174	31	0.79390116	69	TRUE	43
6B	100	50	0.8	0.05	0.03	0.0003	0.78548896	26	0.78864353	89	TRUE	38
6B	100	50	0.7	0.01	0.025	0.003	0.7849632	14	0.78759201	86	TRUE	23
6B	100	50	0.75	0.03	0.02	0.003	0.77539432	5	0.78443743	85	TRUE	14
6B	50	20	0.8	0.05	0.03	0.0003	0.77444795	9	0.78023134	80	TRUE	15
6B	50	20	0.75	0.03	0.02	0.003	0.76319664	4	0.76971609	82	TRUE	9
6B	50	20	0.7	0.01	0.025	0.003	0.75825447	1	0.76656151	84	TRUE	6



## Appendix D

## Dropped Features by Experiment

FEATURE	EXP 1	EXP 2	EXP 3	EXP 4	EXP 5	EXP 6 A/B
ALERT_LIFETIME	X	X	X		X	
COMMITTED_DATE	X	X	X	X	X	X
CVE_ID	X	X	X	X	X	X
CWE_ID	X	X	X	X	X	X
DEST_CREATED_DATE	X	X	X	X	X	X
DEST_FILE_COMPLETE	X	X	X	X	X	X
DEST_FILE_FOLDER	X	X	X	X	X	X
DEST_FILE_FUNCTION_VAR	X	X	X	X	X	X
DEST_FILE_NAME	X	X	X	X	X	X
DEST_FILE_PATH	X	X	X	X	X	X
DEST_LAST_AUTHOR_EMAIL	X	X	X	X	X	X
DEST_LAST_AUTHOR_NAME	X	X	X	X	X	X
DEST_LAST_EDITED_DATE	X	X	X	X	X	X
DEST_UNDRSTD_AvgEssential			X			
DEST_UNDRSTD_CountDeclClass			X			
DEST_UNDRSTD_CountDeclFile	X	X	X	X	X	X
DEST_UNDRSTD_CountLine_Html			X			
DEST_UNDRSTD_CountLineBlank_Html			X			
DEST_UNDRSTD_CountLineComment_Html			X			
DEST_UNDRSTD_MaxInheritanceTree	X	X	X	X	X	X
PROJECT	X	X	X		X	
SOURCE_CREATED_DATE	X	X	X	X	X	X
SOURCE_FILE_COMPLETE	X	X	X	X	X	X
SOURCE_FILE_FOLDER	X	X	X	X	X	X
SOURCE_FILE_FUNCTION_VAR	X	X	X	X	X	X
SOURCE_FILE_NAME	X	X	X	X	X	X
SOURCE_FILE_PATH	X	X	X	X	X	X
SOURCE_LAST_AUTHOR_EMAIL	X	X	X	X	X	X
SOURCE_LAST_AUTHOR_NAME	X	X	X	X	X	X
SOURCE_LAST_EDITED_DATE	X	X	X	X	X	X
SOURCE_UNDRSTD_AvgEssential			X			
SOURCE_UNDRSTD_CountDeclClass			X			
SOURCE_UNDRSTD_CountDeclFile	X	X	X	X	X	X
SOURCE_UNDRSTD_CountLine_Html			X			
SOURCE_UNDRSTD_CountLineBlank_Html			X			
SOURCE_UNDRSTD_CountLineComment_Html			X			
SOURCE_UNDRSTD_MaxInheritanceTree	X	X	X	X	X	X
VERSION_LAST_SEEN	X	X	X	X	X	X
VERSION_START	X	X	X	X	X	X

## Appendix E

## Top Performing Feature Subsets by Experiment

FEATURE	EXP 1	EXP 2	EXP 3	EXP 4	EXP 5	EXP 6A	EXP 6B	TOTAL
ALERT_LIFETIME								0
CATEGORY	X	X	X			X	X	5
CODE_BUG_VULN			X		X	X	X	4
COMMITTED_DATE		X						1
CVE_ID								0
CWE_ID								0
DEST_CREATED_DATE								0
DEST_FILE_AGE	X		X			X	X	4
DEST_FILE_CHURN		X					X	2
DEST_FILE_CLOC	X	X						2
DEST_FILE_COLUMN		X	X	X		X	X	5
DEST_FILE_COMPLETE								0
DEST_FILE_EDIT_FREQUENCY				X			X	2
DEST_FILE_ELOC		X	X	X	X			4
DEST_FILE_EXT						X	X	2
DEST_FILE_FOLDER								0
DEST_FILE_FUNCTION_VAR								0
DEST_FILE_GROWTH					X		X	2
DEST_FILE_LINE	X	X	X	X	X		X	6
DEST_FILE_LINES_ADDED	X		X	X		X		4
DEST_FILE_LINES_DELETED			X					1
DEST_FILE_LOC	X				X	X		3
DEST_FILE_NAME		X						1
DEST_FILE_orig_ccom				X				1
DEST_FILE_orig_ccomdeep			X	X	X			3
DEST_FILE_orig_hvol	X		X		X	X	X	5
DEST_FILE_orig_loc			X	X			X	3
DEST_FILE_orig_nest	X	X				X	X	4
DEST_FILE_orig_nIncomingCalls		X					X	2
DEST_FILE_orig_nIncomingCallsUniq	X			X			X	3
DEST_FILE_orig_nmethods	X		X		X			3
DEST_FILE_orig_nonecholoc	X			X			X	3
DEST_FILE_orig_nOutgoingExternCallsUniq		X		X		X		3
DEST_FILE_orig_nOutgoingExternFlsCalled	X		X	X	X			4
DEST_FILE_orig_nOutgoingExternFlsCalledUniq		X	X					2
DEST_FILE_orig_nOutgoingInternCalls	X					X		2
DEST_FILE_PATH								0
DEST_FILE_PERCENT_MODIFIED	X				X	X	X	4
DEST_FILE_SIZE	X			X	X	X		4
DEST_FILE_STALENESS			X	X	X			3
DEST_FILE_VERSION_ALERT_COUNT	X	X	X		X		X	5
DEST_FILE_VERSION_ALERT_LOC_RATIO								0
DEST_LAST_AUTHOR_EMAIL								0
DEST_LAST_AUTHOR_NAME								0
DEST_LAST_EDITED_DATE								0
DEST_UNDRSTD_AvgCyclomatic				X		X	X	3
DEST_UNDRSTD_AvgCyclomaticModified	X			X	X	X		4
DEST_UNDRSTD_AvgCyclomaticStrict	X							1
DEST_UNDRSTD_AvgEssential	X						X	2
DEST_UNDRSTD_AvgLine	X						X	2
DEST_UNDRSTD_AvgLineBlank	X			X		X	X	4

DEST_UNDRSTD_AvgLineCode			X		X	X		3
DEST_UNDRSTD_AvgLineComment			X	X	X		X	5
DEST_UNDRSTD_CountDeclClass	X				X			2
DEST_UNDRSTD_CountDeclExecutableUnit	X	X	X				X	4
DEST_UNDRSTD_CountDeclFile		X						1
DEST_UNDRSTD_CountDeclFunction		X	X	X			X	4
DEST_UNDRSTD_CountLine		X	X		X	X	X	5
DEST_UNDRSTD_CountLine_Html	X			X				2
DEST_UNDRSTD_CountLine_Javascript	X	X	X	X	X			5
DEST_UNDRSTD_CountLine_Php		X	X		X	X	X	5
DEST_UNDRSTD_CountLineBlank	X	X					X	3
DEST_UNDRSTD_CountLineBlank_Html	X			X	X			3
DEST_UNDRSTD_CountLineBlank_Javascript	X	X		X			X	4
DEST_UNDRSTD_CountLineBlank_Php	X		X		X	X	X	5
DEST_UNDRSTD_CountLineCode	X	X	X				X	4
DEST_UNDRSTD_CountLineCode_Javascript				X	X		X	3
DEST_UNDRSTD_CountLineCode_Php	X			X	X	X		4
DEST_UNDRSTD_CountLineComment		X		X	X	X		4
DEST_UNDRSTD_CountLineComment_Html		X				X	X	3
DEST_UNDRSTD_CountLineComment_Javascript	X				X			2
DEST_UNDRSTD_CountLineComment_Php		X		X			X	3
DEST_UNDRSTD_CountPath				X				1
DEST_UNDRSTD_CountPathLog	X	X	X	X	X			5
DEST_UNDRSTD_CountStmnt	X	X	X		X	X		5
DEST_UNDRSTD_CountStmntDecl	X	X	X				X	4
DEST_UNDRSTD_CountStmntDecl_Javascript			X		X	X		3
DEST_UNDRSTD_CountStmntDecl_Php	X			X			X	3
DEST_UNDRSTD_CountStmntExe		X	X		X			3
DEST_UNDRSTD_CountStmntExe_Javascript	X	X				X	X	4
DEST_UNDRSTD_CountStmntExe_Php	X	X	X		X			4
DEST_UNDRSTD_Cyclomatic		X		X				2
DEST_UNDRSTD_CyclomaticModified	X				X			2
DEST_UNDRSTD_CyclomaticStrict	X			X				2
DEST_UNDRSTD_Essential			X		X		X	3
DEST_UNDRSTD_MaxCyclomatic	X		X	X		X	X	5
DEST_UNDRSTD_MaxCyclomaticModified		X		X	X		X	4
DEST_UNDRSTD_MaxEssential	X			X		X		3
DEST_UNDRSTD_MaxInheritanceTree								0
DEST_UNDRSTD_MaxNesting	X			X				2
DEST_UNDRSTD_RatioCommentToCode			X					1
DEST_UNDRSTD_SumCyclomatic			X	X		X		3
DEST_UNDRSTD_SumCyclomaticModified	X	X	X	X		X	X	6
DEST_UNDRSTD_SumCyclomaticStrict	X		X	X				3
DEST_UNDRSTD_SumEssential			X		X		X	3
FILES_IN_PATH		X			X			2
IN_OWASP_2013		X	X	X	X	X	X	6
IN_OWASP_2017	X	X				X	X	4
LANGUAGE			X		X			2
MATCHED_CVE	X	X	X	X	X	X	X	7
NUM_PATH_HOPS	X	X				X	X	4
OWASP_TOP_TEN_2013	X			X				2
OWASP_TOP_TEN_2017	X							1
PRIORITY					X	X	X	3
PROJECT				X			X	2
SOURCE_CREATED_DATE								0
SOURCE_DEST_SAME_FILE	X		X		X		X	4
SOURCE_FILE_AGE	X	X	X				X	4
SOURCE_FILE_CHURN				X		X		2

SOURCE_FILE_CLOC	X		X				X	3
SOURCE_FILE_COLUMN	X		X	X	X		X	5
SOURCE_FILE_COMPLETE								0
SOURCE_FILE_EDIT_FREQUENCY		X			X	X		3
SOURCE_FILE_ELOC	X	X					X	3
SOURCE_FILE_EXT	X	X	X				X	4
SOURCE_FILE_FOLDER								0
SOURCE_FILE_FUNCTION_VAR								0
SOURCE_FILE_GROWTH		X		X	X	X	X	5
SOURCE_FILE_LINE	X	X					X	3
SOURCE_FILE_LINES_ADDED	X	X	X					3
SOURCE_FILE_LINES_DELETED			X					1
SOURCE_FILE_LOC	X		X	X			X	4
SOURCE_FILE_NAME								0
SOURCE_FILE_orig_ccom			X		X			2
SOURCE_FILE_orig_ccomdeep	X	X	X			X		4
SOURCE_FILE_orig_hvol				X				1
SOURCE_FILE_orig_loc	X	X	X					3
SOURCE_FILE_orig_nest		X	X					2
SOURCE_FILE_orig_nIncomingCalls	X		X					2
SOURCE_FILE_orig_nIncomingCallsUniq			X		X		X	3
SOURCE_FILE_orig_nmethods	X			X		X	X	4
SOURCE_FILE_orig_nonecholoc		X			X	X	X	4
SOURCE_FILE_orig_nOutgoingExternCallsUniq			X		X		X	3
SOURCE_FILE_orig_nOutgoingExternFlsCalled	X	X			X		X	4
SOURCE_FILE_orig_nOutgoingExternFlsCalledUniq		X			X		X	3
SOURCE_FILE_orig_nOutgoingInternCalls				X	X	X	X	4
SOURCE_FILE_PATH								0
SOURCE_FILE_PERCENT_MODIFIED						X		1
SOURCE_FILE_SIZE	X		X		X		X	4
SOURCE_FILE_STALENESS	X	X	X		X	X		5
SOURCE_FILE_VERSION_ALERT_COUNT	X	X	X					3
SOURCE_FILE_VERSION_ALERT_LOC_RATIO	X			X		X		3
SOURCE_LAST_AUTHOR_EMAIL								0
SOURCE_LAST_AUTHOR_NAME								0
SOURCE_LAST_EDITED_DATE								0
SOURCE_UNDRSTD_AvgCyclomatic		X		X			X	3
SOURCE_UNDRSTD_AvgCyclomaticModified							X	1
SOURCE_UNDRSTD_AvgCyclomaticStrict			X	X	X			3
SOURCE_UNDRSTD_AvgEssential	X	X		X	X		X	5
SOURCE_UNDRSTD_AvgLine	X	X		X			X	4
SOURCE_UNDRSTD_AvgLineBlank	X	X	X	X	X		X	6
SOURCE_UNDRSTD_AvgLineCode	X	X	X	X	X		X	6
SOURCE_UNDRSTD_AvgLineComment		X		X	X	X	X	5
SOURCE_UNDRSTD_CountDeclClass		X		X		X	X	4
SOURCE_UNDRSTD_CountDeclExecutableUnit			X				X	2
SOURCE_UNDRSTD_CountDeclFile								0
SOURCE_UNDRSTD_CountDeclFunction			X	X		X		3
SOURCE_UNDRSTD_CountLine	X		X	X	X	X	X	6
SOURCE_UNDRSTD_CountLine_Html				X	X			2
SOURCE_UNDRSTD_CountLine_Javascript	X	X		X	X			4
SOURCE_UNDRSTD_CountLine_Php		X	X		X	X		4
SOURCE_UNDRSTD_CountLineBlank	X		X	X		X		4
SOURCE_UNDRSTD_CountLineBlank_Html							X	1
SOURCE_UNDRSTD_CountLineBlank_Javascript	X	X	X		X		X	5
SOURCE_UNDRSTD_CountLineBlank_Php	X		X			X		3
SOURCE_UNDRSTD_CountLineCode					X			1
SOURCE_UNDRSTD_CountLineCode_Javascript		X	X		X	X		4

SOURCE_UNDRSTD_CountLineCode_Php	X	X	X		X			4
SOURCE_UNDRSTD_CountLineComment	X			X	X	X		4
SOURCE_UNDRSTD_CountLineComment_Html				X	X			2
SOURCE_UNDRSTD_CountLineComment_Javascript	X	X	X	X				4
SOURCE_UNDRSTD_CountLineComment_Php					X	X	X	3
SOURCE_UNDRSTD_CountPath			X					1
SOURCE_UNDRSTD_CountPathLog	X		X	X	X			4
SOURCE_UNDRSTD_CountStmt	X		X					2
SOURCE_UNDRSTD_CountStmtDecl		X	X	X	X	X	X	6
SOURCE_UNDRSTD_CountStmtDecl_Javascript	X		X	X	X	X	X	6
SOURCE_UNDRSTD_CountStmtDecl_Php					X		X	2
SOURCE_UNDRSTD_CountStmtExe	X					X		2
SOURCE_UNDRSTD_CountStmtExe_Javascript		X	X	X			X	4
SOURCE_UNDRSTD_CountStmtExe_Php							X	1
SOURCE_UNDRSTD_Cyclomatic	X		X	X				3
SOURCE_UNDRSTD_CyclomaticModified	X		X	X	X			4
SOURCE_UNDRSTD_CyclomaticStrict	X			X				2
SOURCE_UNDRSTD_Essential	X	X	X	X				4
SOURCE_UNDRSTD_MaxCyclomatic		X				X	X	3
SOURCE_UNDRSTD_MaxCyclomaticModified	X	X	X				X	4
SOURCE_UNDRSTD_MaxEssential	X			X			X	3
SOURCE_UNDRSTD_MaxInheritanceTree								0
SOURCE_UNDRSTD_MaxNesting		X						1
SOURCE_UNDRSTD_RatioCommentToCode		X	X	X	X			4
SOURCE_UNDRSTD_SumCyclomatic	X		X			X		3
SOURCE_UNDRSTD_SumCyclomaticModified	X	X	X				X	4
SOURCE_UNDRSTD_SumCyclomaticStrict	X			X		X		3
SOURCE_UNDRSTD_SumEssential		X				X		2
TOOL		X				X		2
TYPE			X	X	X			3
VERSION_ALERT_COUNT	X	X				X	X	4
VERSION_LAST_SEEN								0
VERSION_START								0

## Appendix F

### Sonar Scan Script

```
#####
#AUTHOR: KATHY GOESCHEL
#DATE 09/02/2019
#PURPOSE: PHD DISSERTATION - NOVA SOUTHEASTERN UNIVERSITY
#GENERAL ML MODEL - SVM ONLY - FOR STATIC ANALYSIS CLASSIFICATION IMPROVEMENTS
#####
#python sonar-scan.py

import os, sys, logging, time, csv, datetime
from git import Repo
from git import Git
from subprocess import PIPE
import subprocess

#TO USE -- MAKE SURE THE SONAR SERVER IS RUNNING ON LOCALHOST
#1. go to cmd line to start server (/sonarqube-6.7.1/bin/macosx-universal-64/sonar.sh console)
#2. go the new cmd line (export PATH=$PATH:/DATASET/sonar-scanner-3.3.0.1492-macosx/bin)
#3. workon [specify virtualenv]
#4. Run script with arg of project name (sonar-scan.py PHPMYADMIN)

projectSpecified = str(sys.argv[1])

now = datetime.datetime.now()
nowFormatted = now.strftime("%Y-%m-%d_%H_%M")

dirRoot = os.path.abspath(os.path.join(os.path.dirname( __file__ ), '..'))
logging.basicConfig(filename=dirRoot+'/logs/SonarScanLog-'+nowFormatted+'.log',
filemode='w',level=logging.DEBUG)
inputFile = dirRoot + "/toScan.csv"
scanList = []
repoPath = ""

def getProjectsFromCSV():
    logging.info("IMPORTING CSV")
    with open(inputFile, mode='rb') as csv_file:
        csv_reader = csv.DictReader(csv_file)
        for line in csv_reader:
            scanList.append(line)
    return scanList

try:
    getProjectsFromCSV()

    for i in scanList:

        #skip ones that we dont have a tag for
        if i['TAG'] == 'NA':
            continue

        #skip ones that are not the project we specified
        if i['PROJECT'] != projectSpecified:
            continue

        logging.info("Scanning " + i['PROJECT'] + " " + i['COMMIT_HASH'])

        if i['PROJECT'] == 'DRUPAL':
            repoPath = "/SOURCE_CODE/DRUPAL/drupal/"
        elif i['PROJECT'] == 'MOODLE':
            repoPath = "/SOURCE_CODE/MOODLE/moodle/"
        elif i['PROJECT'] == 'PHPMYADMIN':
            repoPath = "/SOURCE_CODE/PHPMYADMIN/phpmyadmin/"
        else:
            logging.error("repo location problem")
            sys.exit(1)

        #GIT CHECKOUT HASH VERSION
        try:
            g = Git(repoPath)
```

```

        g.checkout(i['COMMIT_HASH'])
    except:
        logging.error("ERROR: Changing Repo Checkout Hash")
        continue

    #START SCAN
    try:
        retcode = subprocess.call("echo PASSWORD|sudo -S sonar-scanner -Dsonar.projectKey=" +
i['PROJECT'] + "_" + i['TAG'] + " -Dproject.projectName=" + i['PROJECT'] + "_" + i['TAG'] + " -
Dsonar.projectVersion=" + i['TAG'] + " -Dsonar.projectBaseDir=" + repoPath + " -Dsonar.sources=", shell=True)
        if retcode < 0:
            logging.debug("\tMSG: was terminated " + str(retcode))
        else:
            if retcode == 0:
                logging.debug("\tMSG: Success - scan returned " + str(retcode))
            else:
                logging.error("\tERROR: Failure - scan returned " + str(retcode))
    except OSError as e:
        logging.error("\tERROR: Scan execution failed: " + e)

    #GET REPORT
    try:
        num = 1
        maxPage = 500
        r = requests.get('http://localhost:9000/api/issues/search?componentKeys=' + i['PROJECT'] + '_' +
i['TAG'] + '&pageIndex=' + str(num) + '&pageSize=-1&types=BUG%2CVULNERABILITY', allow_redirects=True)
        temp = json.loads(r.content)
        totalIssues = int(temp['total'])
        if totalIssues > 10000:
            print('ALERT: more issues than possible to grab...' + i['PROJECT'] + '_' + i['TAG'])
            continue
        open(dirRoot + '/SCAN_RESULTS/Sonar' + i["PROJECT"] + '_' + i["TAG"] + '_' + str(num) + '.json',
'wb').write(r.content)
        while num < ((totalIssues/maxPage)+1):
            logging.info(str(num))
            num += 1
            s = requests.get('http://localhost:9000/api/issues/search?componentKeys=' + i['PROJECT'] +
 '_' + i['TAG'] + '&pageIndex=' + str(num) + '&pageSize=-1&types=BUG%2CVULNERABILITY', allow_redirects=True)
            temp2 = json.loads(s.content)
            open(dirRoot + '/SCAN_RESULTS/Sonar' + i["PROJECT"] + '_' + i["TAG"] + '_' + str(num) +
'.json', 'wb').write(s.content)
            time.sleep(2)
    except OSError as e:
        logging.error("\tERROR: Failed to get issues page: " + e)

except:
    logging.error("ERROR: Overall")
    sys.exit(1)

```

## Appendix G

### Data Pre-Processing

```
#####
#AUTHOR: KATHY GOESCHEL
#DATE 09/02/2019
#PURPOSE: PHD DISSERTATION - NOVA SOUTHEASTERN UNIVERSITY
#GENERAL ML MODEL FOR STATIC ANALYSIS CLASSIFICATION IMPROVEMENTS
#####

from datetime import datetime
import csv
import logging
import os
import sys
import time

import numpy as np
import pandas as pd
from random import random, randint

from sklearn.preprocessing import LabelEncoder, LabelBinarizer, OneHotEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split
from sklearn import svm
import sklearn.metrics as metrics

#####DECLARE PATHS / VARIABLES#####

pythonFileUsed = str(sys.argv[0]).split(".")[0]
experimentRun = str(sys.argv[1])
datetimeFormat = '%Y-%m-%d %H:%M:%S.%f'
rightNow = datetime.now().strftime(datetimeFormat)

dataFiles = []
dataFilePreProcessed = "PreProcessed_"+experimentRun+".csv"
dataFilePreProcessedScaled = "PreProcessedScaled_"+experimentRun+".csv"
dataFilePreProcessedFeatures = "PreProcessed_"+experimentRun+"_Features.txt"

#####MAIN PATHS
mainDir = "/MODEL/"

#RAW DATASET
dataDir = "/DATASET/NEW/THE_SET/"

#WHERE TO SAVE THE PREPROCESSED DATA SET (SANS SCALING)
dataDirSave = mainDir + "DATA/"
if not os.path.exists(dataDirSave + pythonFileUsed):
    os.mkdir(dataDirSave + pythonFileUsed)
dataDirSave = dataDirSave + pythonFileUsed + "/"

#####INPUT FILES
dataFilePreProcessed = dataDirSave + dataFilePreProcessed
dataFilePreProcessedScaled = dataDirSave + dataFilePreProcessedScaled
dataFilePreProcessedFeatures = dataDirSave + dataFilePreProcessedFeatures

theFeatures = []
theLabel = "CLASSIFICATION"
numFeatures = 0

drupalversions = [
'6.0','6.1','6.2','6.3','6.4','6.5','6.6','6.7','6.10','6.11','6.12','6.13','6.14','6.15','6.16','6.17','6.18',
'6.19','6.20','6.22','6.23','6.24','6.26','6.29','6.31','6.33','6.34','6.35','6.37','6.38']

phpversions = [
'RELEASE_2_2_0','RELEASE_2_2_1','RELEASE_2_2_2','RELEASE_2_2_3','RELEASE_2_2_4','RELEASE_2_2_5','RELEASE_2_2_6',
'RELEASE_2_3_0','RELEASE_2_3_1','RELEASE_2_3_2','RELEASE_2_3_3PL1','RELEASE_2_4_0','RELEASE_2_5_0','RELEASE_2_5_1',
'RELEASE_2_5_2','RELEASE_2_5_4','RELEASE_2_5_5PL1','RELEASE_2_5_6','RELEASE_2_6_1PL3','RELEASE_2_6_2PL1',
'RELEASE_2_6_3PL1','RELEASE_2_6_4PL4','RELEASE_2_7_0PL2','RELEASE_2_8_1','RELEASE_2_9_0','RELEASE_2_9_1_1','REL',
'EASE_2_9_2','RELEASE_2_10_0','RELEASE_2_10_1RC1','RELEASE_2_10_2','RELEASE_2_10_3','RELEASE_2_11_0','RELEASE_2_11_1',
'RELEASE_2_11_2','RELEASE_2_11_3','RELEASE_2_11_4','RELEASE_2_11_5','RELEASE_2_11_6','RELEASE_2_11_7','RE',
'LEASE_2_11_8','RELEASE_2_11_9','RELEASE_3_0_0','RELEASE_3_0_1','RELEASE_3_1_0','RELEASE_3_1_1','RELEASE_3_1_2',
'RELEASE_3_1_3','RELEASE_3_1_4','RELEASE_3_1_5','RELEASE_3_2_0','RELEASE_3_2_2','RELEASE_3_2_3','RELEASE_3_2_4',
'RELEASE_3_2_5','RELEASE_3_3_0','RELEASE_3_3_1','RELEASE_3_3_2','RELEASE_3_3_3','RELEASE_3_3_4','RELEASE_3_3_5']
```



```

', 'RELEASE_3_3_6', 'RELEASE_3_3_7', 'RELEASE_3_3_8', 'RELEASE_3_3_9', 'RELEASE_3_4_0', 'RELEASE_3_4_1', 'RELEASE_3_4_2', 'RELEASE_3_4_3', 'RELEASE_3_4_4', 'RELEASE_3_4_5', 'RELEASE_3_4_6', 'RELEASE_3_4_7', 'RELEASE_3_4_8', 'RELEASE_3_4_9', 'RELEASE_3_5_0', 'RELEASE_3_5_1', 'RELEASE_3_5_2', 'RELEASE_3_5_3', 'RELEASE_3_5_4', 'RELEASE_3_5_5', 'RELEASE_3_5_6', 'RELEASE_3_5_7', 'RELEASE_3_5_8', 'RELEASE_4_0_0', 'RELEASE_4_0_1', 'RELEASE_4_0_2', 'RELEASE_4_0_3', 'RELEASE_4_0_4', 'RELEASE_4_0_5', 'RELEASE_4_0_6', 'RELEASE_4_0_7', 'RELEASE_4_0_8']
moodLeversions =
['v1.0.0', 'v1.0.1', 'v1.0.2', 'v1.0.3', 'v1.0.4', 'v1.0.5', 'v1.0.6', 'v1.0.7', 'v1.0.8', 'v1.0.9', 'v1.1.0', 'v1.1.1', 'v1.2.0', 'v1.2.1', 'v1.3.0', 'v1.3.1', 'v1.3.2', 'v1.3.3', 'v1.3.4', 'v1.4.0', 'v1.4.1', 'v1.4.2', 'v1.4.3', 'v1.4.4', 'v1.4.5', 'v1.5.0', 'v1.5.1', 'v1.5.2', 'v1.5.3', 'v1.6.0', 'v1.6.1', 'v1.6.2', 'v1.6.3', 'v1.7.0', 'v1.7.1', 'v1.7.2', 'v1.8.0', 'v1.8.1', 'v1.8.2', 'v1.8.3', 'v1.8.4', 'v1.9.0', 'v1.9.1', 'v1.9.2', 'v2.0.0', 'v2.0.1', 'v2.0.2', 'v2.0.3', 'v2.1.0', 'v2.1.1', 'v2.1.2', 'v2.1.3', 'v2.2.0', 'v2.2.1', 'v2.2.2', 'v2.2.3', 'v2.3.0', 'v2.3.1', 'v2.3.2', 'v2.3.3', 'v2.4.0', 'v2.4.1', 'v2.4.2', 'v2.4.3', 'v2.4.4', 'v2.5.0', 'v2.5.1', 'v2.5.2', 'v2.5.3', 'v2.6.0']

#####PRE-PROCESSING DECLARATIONS
featuresToOneHot =
["PROJECT", "TOOL", "CATEGORY", "OWASP_TOP_TEN_2013", "OWASP_TOP_TEN_2017", "TYPE", "CODE_BUG_VULN", "LANGUAGE", "SOURCE_FILE_EXT", "DEST_FILE_EXT"]
featuresToCatEncode = []
featuresBoolean = ["IN_OWASP_2013", "IN_OWASP_2017", "SOURCE_DEST_SAME_FILE", "MATCHED_CVE"]
featuresToDrop =
["VERSION_START", "VERSION_LAST_SEEN", "COMMITTED_DATE", "CWE_ID", "SOURCE_FILE_COMPLETE", "SOURCE_FILE_PATH", "SOURCE_FILE_FOLDER", "SOURCE_FILE_NAME", "SOURCE_FILE_FUNCTION_VAR", "DEST_FILE_COMPLETE", "DEST_FILE_PATH", "DEST_FILE_FOLDER", "DEST_FILE_NAME", "DEST_FILE_FUNCTION_VAR", "SOURCE_CREATED_DATE", "SOURCE_LAST_EDITED_DATE", "SOURCE_LAST_AUTHOR_NAME", "SOURCE_LAST_AUTHOR_EMAIL", "DEST_CREATED_DATE", "DEST_LAST_EDITED_DATE", "DEST_LAST_AUTHOR_NAME", "DEST_LAST_AUTHOR_EMAIL", "CVE_ID"]
featuresDropAfterEncoding =
["OWASP_TOP_TEN_2017_NONE", "OWASP_TOP_TEN_2013_NONE", "SOURCE_FILE_EXT_NONE", "DEST_FILE_EXT_NONE", "CATEGORY_NONE"]

featuresNumericalScaleNoBlanks = ["ALERT_LIFETIME", "NUM_PATH_HOPS", "FILES_IN_PATH", "VERSION_ALERT_COUNT"]
featuresNumericalScaleHasBlanks =
["SOURCE_FILE_LINE", "SOURCE_FILE_COLUMN", "DEST_FILE_LINE", "DEST_FILE_COLUMN", "SOURCE_FILE_SIZE", "SOURCE_FILE_LOC", "SOURCE_FILE_CLOC", "SOURCE_FILE_ELOC", "SOURCE_FILE_AGE", "SOURCE_FILE_STALENESS", "DEST_FILE_SIZE", "DEST_FILE_LOC", "DEST_FILE_CLOC", "DEST_FILE_ELOC", "DEST_FILE_AGE", "DEST_FILE_STALENESS", "SOURCE_UNDRSTD_AvgCyclomatic", "SOURCE_UNDRSTD_AvgCyclomaticModified", "SOURCE_UNDRSTD_AvgCyclomaticStrict", "SOURCE_UNDRSTD_AvgEssential", "SOURCE_UNDRSTD_AvgLine", "SOURCE_UNDRSTD_AvgLineBlank", "SOURCE_UNDRSTD_AvgLineCode", "SOURCE_UNDRSTD_AvgLineComment", "SOURCE_UNDRSTD_CountDeclClass", "SOURCE_UNDRSTD_CountDeclExecutableUnit", "SOURCE_UNDRSTD_CountDeclFile", "SOURCE_UNDRSTD_CountDeclFunction", "SOURCE_UNDRSTD_CountLine", "SOURCE_UNDRSTD_CountLineBlank", "SOURCE_UNDRSTD_CountLineBlankHtml", "SOURCE_UNDRSTD_CountLineBlankJavascript", "SOURCE_UNDRSTD_CountLineBlankPhp", "SOURCE_UNDRSTD_CountLineCode", "SOURCE_UNDRSTD_CountLineCodeJavascript", "SOURCE_UNDRSTD_CountLineCodePhp", "SOURCE_UNDRSTD_CountLineComment", "SOURCE_UNDRSTD_CountLineCommentHtml", "SOURCE_UNDRSTD_CountLineCommentJavascript", "SOURCE_UNDRSTD_CountLineCommentPhp", "SOURCE_UNDRSTD_CountLineHtml", "SOURCE_UNDRSTD_CountLineJavascript", "SOURCE_UNDRSTD_CountLinePhp", "SOURCE_UNDRSTD_CountPath", "SOURCE_UNDRSTD_CountPathLog", "SOURCE_UNDRSTD_CountStmt", "SOURCE_UNDRSTD_CountStmtDecl", "SOURCE_UNDRSTD_CountStmtDeclJavascript", "SOURCE_UNDRSTD_CountStmtDeclPhp", "SOURCE_UNDRSTD_CountStmtExe", "SOURCE_UNDRSTD_CountStmtExeJavascript", "SOURCE_UNDRSTD_CountStmtExePhp", "SOURCE_UNDRSTD_Cyclomatic", "SOURCE_UNDRSTD_CyclomaticModified", "SOURCE_UNDRSTD_CyclomaticStrict", "SOURCE_UNDRSTD_Essential", "SOURCE_UNDRSTD_MaxCyclomatic", "SOURCE_UNDRSTD_MaxCyclomaticModified", "SOURCE_UNDRSTD_MaxEssential", "SOURCE_UNDRSTD_MaxInheritanceTree", "SOURCE_UNDRSTD_MaxNesting", "SOURCE_UNDRSTD_RatioCommentToCode", "SOURCE_UNDRSTD_SumCyclomatic", "SOURCE_UNDRSTD_SumCyclomaticModified", "SOURCE_UNDRSTD_SumCyclomaticStrict", "SOURCE_UNDRSTD_SumEssential", "DEST_UNDRSTD_AvgCyclomatic", "DEST_UNDRSTD_AvgCyclomaticModified", "DEST_UNDRSTD_AvgCyclomaticStrict", "DEST_UNDRSTD_AvgEssential", "DEST_UNDRSTD_AvgLine", "DEST_UNDRSTD_AvgLineBlank", "DEST_UNDRSTD_AvgLineCode", "DEST_UNDRSTD_AvgLineComment", "DEST_UNDRSTD_CountDeclClass", "DEST_UNDRSTD_CountDeclExecutableUnit", "DEST_UNDRSTD_CountDeclFile", "DEST_UNDRSTD_CountDeclFunction", "DEST_UNDRSTD_CountLine", "DEST_UNDRSTD_CountLineBlank", "DEST_UNDRSTD_CountLineBlankHtml", "DEST_UNDRSTD_CountLineBlankJavascript", "DEST_UNDRSTD_CountLineBlankPhp", "DEST_UNDRSTD_CountLineCode", "DEST_UNDRSTD_CountLineCodeJavascript", "DEST_UNDRSTD_CountLineCodePhp", "DEST_UNDRSTD_CountLineComment", "DEST_UNDRSTD_CountLineCommentHtml", "DEST_UNDRSTD_CountLineCommentJavascript", "DEST_UNDRSTD_CountLineCommentPhp", "DEST_UNDRSTD_CountLineHtml", "DEST_UNDRSTD_CountLineJavascript", "DEST_UNDRSTD_CountLinePhp", "DEST_UNDRSTD_CountPath", "DEST_UNDRSTD_CountPathLog", "DEST_UNDRSTD_CountStmt", "DEST_UNDRSTD_CountStmtDecl", "DEST_UNDRSTD_CountStmtDeclJavascript", "DEST_UNDRSTD_CountStmtDeclPhp", "DEST_UNDRSTD_CountStmtExe", "DEST_UNDRSTD_CountStmtExeJavascript", "DEST_UNDRSTD_CountStmtExePhp", "DEST_UNDRSTD_Cyclomatic", "DEST_UNDRSTD_CyclomaticModified", "DEST_UNDRSTD_CyclomaticStrict", "DEST_UNDRSTD_Essential", "DEST_UNDRSTD_MaxCyclomatic", "DEST_UNDRSTD_MaxCyclomaticModified", "DEST_UNDRSTD_MaxEssential", "DEST_UNDRSTD_MaxInheritanceTree", "DEST_UNDRSTD_MaxNesting", "DEST_UNDRSTD_RatioCommentToCode", "DEST_UNDRSTD_SumCyclomatic", "DEST_UNDRSTD_SumCyclomaticModified", "DEST_UNDRSTD_SumCyclomaticStrict", "DEST_UNDRSTD_SumEssential", "SOURCE_FILE_VERSION_ALERT_COUNT", "SOURCE_FILE_VERSION_ALERT_LOC_RATIO", "DEST_FILE_VERSION_ALERT_COUNT", "DEST_FILE_VERSION_ALERT_LOC_RATIO", "SOURCE_FILE_LINES_ADDED", "SOURCE_FILE_LINES_DELETED", "SOURCE_FILE_CHURN", "SOURCE_FILE_GROWTH", "SOURCE_FILE_PERCENT_MODIFIED", "SOURCE_FILE_EDIT_FREQUENCY", "DEST_FILE_LINES_ADDED", "DEST_FILE_LINES_DELETED", "DEST_FILE_CHURN", "DEST_FILE_GROWTH", "DEST_FILE_PERCENT_MODIFIED", "DEST_FILE_EDIT_FREQUENCY"]
featuresNumericalScaleHasBlanksIsSparse =
["SOURCE_FILE_orig_nonecholoc", "SOURCE_FILE_orig_loc", "SOURCE_FILE_orig_nmethods", "SOURCE_FILE_orig_ccomdeep", "SOURCE_FILE_orig_ccom", "SOURCE_FILE_orig_nest", "SOURCE_FILE_orig_hvol", "SOURCE_FILE_orig_nIncomingCalls", "SOURCE_FILE_orig_nIncomingCallsUniq", "SOURCE_FILE_orig_nOutgoingInternCalls", "SOURCE_FILE_orig_nOutgoingExternFlsCalled", "SOURCE_FILE_orig_nOutgoingExternFlsCalledUniq", "SOURCE_FILE_orig_nOutgoingExternCallsUniq", "DEST_FILE_orig_nonecholoc", "DEST_FILE_orig_loc", "DEST_FILE_orig_nmethods", "DEST_FILE_orig_ccomdeep", "DEST_FILE_orig_ccom", "DEST_FILE_orig_nest", "DEST_FILE_orig_hvol", "DEST_FILE_orig_nIncomingCalls", "DEST_FILE_orig_nIncomingCallsUniq", "DEST_FILE_orig_nOutgoingInternCalls", "DEST_FILE_orig_nOutgoingExternFlsCalled", "DEST_FILE_orig_nOutgoingExternFlsCalledUniq", "DEST_FILE_orig_nOutgoingExternCallsUniq"]

```

```

featuresNewMapping = []
deletedCols = []

#####FUNCTIONS#####

def loadCSV(file):
    temp = pd.read_csv(file, delimiter=',', skiprows=0).replace('","')
    return temp

def loadMultCSV():
    df_from_each_file = (pd.read_csv(dataDir + f) for f in dataFiles)
    concatenated_df = pd.concat(df_from_each_file, ignore_index=True)
    return concatenated_df

def returnTimeFormatted():
    return datetime.now().strftime(datetimeFormat)

def secondsBetween(d1, d2):
    d1 = datetime.strptime(d1, datetimeFormat)
    d2 = datetime.strptime(d2, datetimeFormat)
    return abs((d2 - d1).seconds)

def dayOfWeek(theDate, dateFormat):
    res = datetime.strptime(theDate, dateFormat)
    return datetime.weekday(res)

def columnHasUniqueValues(column):
    if howManyUniqueValues(column) > 1:
        return True
    return False

def howManyUniqueValues(column):
    return len(np.unique(data[column]))

def trackNewColumnNames(column):
    featuresNewMapping.append({column:[column+"_"+i for i in np.unique(data[column])])})

def encodeLabel(column):
    lb_make = LabelEncoder()
    lb_results = lb_make.fit_transform(data[column])
    return pd.DataFrame(lb_results, columns=[column])

def encodeLabelBinarizer(column):
    lb_style = LabelBinarizer()
    lb_results = lb_style.fit_transform(data[column])
    return pd.DataFrame(lb_results, columns=[column + "_" +str(lb_style.classes_[i]) for i in
range(len(lb_style.classes_))])

#####START#####

#####PREAMBLE
rightNow = returnTimeFormatted()

#GET THE DATA FILES
for i in phpversions:
    dataFiles.append("PHPMYADMIN/PHPMYADMIN_"+i+".csv")
for i in moodleversions:
    dataFiles.append("MOODLE/MOODLE_"+i+".csv")
for i in drupalversions:
    dataFiles.append("DRUPAL/DRUPAL_"+i+".csv")
print(dataFiles)

#####LOAD / GET DATA SET
data = loadMultCSV()

#GET THE COLUMN HEADERS / FEATURE NAMES
for i in data.columns:
    #DONT TAKE THE CLASSIFICATION COLUMN
    if i != theLabel:
        theFeatures.append(i)
numFeatures = len(theFeatures)

#####DATA PRE-PROCESSING

#DROP CLASSIFICATION OF DELETED.....

```

```

data = data[data.CLASSIFICATION != 'DELETED']

#DROP DUPLICATE ROWS
considerDups =
["PROJECT","TOOL","PRIORITY","CATEGORY","CWE_ID","IN_OWASP_2013","IN_OWASP_2017","OWASP_TOP_TEN_2013","OWASP_TO
P_TEN_2017","TYPE","CODE_BUG_VULN","LANGUAGE","SOURCE_FILE_COMPLETE","SOURCE_FILE_LINE","SOURCE_FILE_COLUMN","S
OURCE_FILE_FUNCTION_VAR","DEST_FILE_COMPLETE","DEST_FILE_LINE","DEST_FILE_COLUMN","DEST_FILE_FUNCTION_VAR"]
t = len(data)
data.drop_duplicates(subset=considerDups, keep='first',inplace=True)

#KEEP SAME NUMBER OF EACH CLASS (DATA SET IS HEAVILY SKEWED)
numActionable = len(data[data['CLASSIFICATION'] == 'ACTIONABLE'])
numRows = len(data)
numUnActionable = numRows - numActionable

if (numActionable / numRows) < .45:
    numRowsToDrop = (numRows - numActionable) - numActionable
    data.drop(data[data['CLASSIFICATION'] == 'UNACTIONABLE'].sample(n=numRowsToDrop).index,inplace=True)
elif (numUnActionable / numRows) < .45:
    numRowsToDrop = (numRows - numUnActionable) - numUnActionable
    data.drop(data[data['CLASSIFICATION'] == 'ACTIONABLE'].sample(n=numRowsToDrop).index,inplace=True)

#GET THE COLUMN HEADERS / FEATURE NAMES
for i in data.columns:
    #DONT INCLUDE THE CLASSIFICATION COLUMN
    if i != theLabel:
        theFeatures.append(i)
numFeatures = len(theFeatures)

#DROP FEATURES SPECIFIED.....
for i in featuresToDrop:
    data = data.drop(columns=i)
    deletedCols.append(i)

#DROP COLUMNS WITH NO DATA AT ALL
colsBeforeDrop = data.columns
data.dropna(axis=1, how='all', inplace=True)
colsAfterDrop = data.columns
#TRACK WHICH COLUMNS WERE DELETED
for i in colsBeforeDrop:
    if i not in colsAfterDrop:
        deletedCols.append(i)

#BOOLEAN
for i in featuresBoolean:
    if i in data:
        data[i] = data[i].astype(int)

#FILL SOME MISSING DATA BEFORE LABEL ENCODING
for i in featuresToOneHot:
    if i in data:
        data[i] = data[i].fillna(value="None")

for i in featuresToCatEncode:
    if i in data:
        data[i] = data[i].fillna(value="None")

#DROP COLUMN IF ALL DATA IN THAT COLUMN IS THE SAME
for i in theFeatures:
    if i in data:
        if not columnHasUniqueValues(i):
            data = data.drop(columns=i)
            deletedCols.append(i)

#CAPITILIZATION CONSISTENCY FOR VALUES
for i in featuresToOneHot:
    if i in data:
        data[i] = data[i].str.upper()

for i in featuresToCatEncode:
    if i in data:
        data[i] = data[i].str.upper()

for i in featuresToOneHot:
    if i in data:
        if columnHasUniqueValues(i):

```

```

    trackNewColumnNames(i)
    res = encodeLabelBinarizer(i)
    data = data.drop(columns=i) #this is really a replace
    data = pd.concat([data, res], axis=1)

for i in featuresToCatEncode:
    if i in data:
        if columnHasUniqueValues(i):
            res = encodeLabel(i)
            data = data.drop(columns=i) #this is really a replace
            data = pd.concat([data, res], axis=1)

for i in featuresDropAfterEncoding:
    if i in data:
        data = data.drop(columns=i)
        for n in featuresNewMapping:
            for m in n:
                if m == i:
                    m.remove(i)

#FILL MISSING NUMERICAL DATA WITH MEDIAN OR MODES BASED ON ANALYSIS
for i in featuresNumericalScaleHasBlanks:
    if i in data:
        data[i].fillna(data[i].median(), inplace=True)

for i in featuresNumericalScaleNoBlanks:
    if i in data:
        data[i].fillna(data[i].median(), inplace=True)

for i in featuresNumericalScaleHasBlanksIsSparse:
    if i in data:
        if i == 'SOURCE_FILE_orig_hvol' or i == 'DEST_FILE_orig_hvol':
            data[i].fillna(data[i].median(), inplace=True)
        else:
            data[i].fillna(data[i].mode()[0], inplace=True)

#AFTER PROCESSING DONE.....DROP COLUMN IF ALL DATA IN THAT COLUMN IS THE SAME
for i in theFeatures:
    if i in data:
        if not columnHasUniqueValues(i):
            data = data.drop(columns=i)
            deletedCols.append(i)

for i in deletedCols:
    if i in theFeatures:
        theFeatures.remove(i)

data.to_csv(dataFilePreProcessed,index=False)

with open(dataFilePreProcessedFeatures,"a+") as f:
    f.write("DELETED COLS: \r\n%s" %deletedCols)
    f.write("\r\n\r\nNEW FEATURES MAPPING: \r\n%s" %featuresNewMapping)
    f.write("\r\nREMOVE THE FOLLOWING BEFORE MODEL: \r\n%s" %featuresDropAfterEncoding)
    f.write("\r\n\r\nRAW DATA FILES USED FOR THIS DATASET...\r\n%s" %dataFiles)
    f.write("\r\n\r\n%s" %printNote)

exit()

```

## Appendix H

### Control Classifier – Model A

```
#####
#AUTHOR: KATHY GOESCHEL
#DATE 09/02/2019
#PURPOSE: PHD DISSERTATION - NOVA SOUTHEASTERN UNIVERSITY
#GENERAL ML MODEL - SVM ONLY - FOR STATIC ANALYSIS CLASSIFICATION IMPROVEMENTS
#####
#python MODEL_A.py
#THIS FILE WILL LOOP THROUGH THE EXPERIMENTS AND TRAIN AND TEST THE EXPERIMENT FILES FOR ALL SVMs SPECIFIED

from datetime import datetime
import csv
import os
import sys
import time

import numpy as np
import pandas as pd
from random import random, randint

from sklearn.preprocessing import RobustScaler
from sklearn import svm
from sklearn.externals import joblib
import sklearn.metrics as metrics

#####DECLARE PATHS / VARIABLES#####

kernel = ''
pythonFileUsed = str(sys.argv[0]).split(".")[0]
experimentRun = ['Exp1', 'Exp2', 'Exp3', 'Exp4', 'Exp5', 'Exp6a', 'Exp6b']
trainOrTest = ['TRAIN', 'TEST']
mod = ['BasicLinear']

datetimeFormat = '%Y-%m-%d %H:%M:%S.%f'

note = "SVM MODEL A - SVM ONLY - USING PREPROCESSED, SCALED DATA"

#####MAIN PATHS
mainDir = "/MODEL/"

#####LOAD CSV
def loadCSV(file):
    temp = pd.read_csv(file, delimiter=',', skiprows=0).replace('","')
    return temp

def returnTimeFormatted():
    return datetime.now().strftime(datetimeFormat)

def secondsBetween(d1, d2):
    d1 = datetime.strptime(d1, datetimeFormat)
    d2 = datetime.strptime(d2, datetimeFormat)
    return abs((d2 - d1).seconds)

#####PREAMBLE
print(note)

for e in experimentRun:
    for t in trainOrTest:
        for m in mod:

            start = returnTimeFormatted()
            data = ''

            dataFile = mainDir + "DATA/MODEL_PRE-PROCESS/PreProcessed " + e + "_" + t + ".csv"
            modelFile = mainDir + "RESULTS/" + pythonFileUsed + "/TRAINED_MODELS/" + e + "_" + m + "_MODEL.sav"
            scalerFile = mainDir + "RESULTS/" + pythonFileUsed + "/TRAINED_MODELS/" + e + "_" + m + "_SCALER.sav"

            #####RESULTS FILE
            resultsDir = mainDir + "RESULTS/"
            if not os.path.exists(resultsDir + pythonFileUsed):
```

```

os.mkdir(resultsDir + pythonFileUsed)
resultsDir = mainDir + "RESULTS/" + pythonFileUsed + "/"
resultsFile = resultsDir + pythonFileUsed + "_" + e + "_" + m + "_" + t + ".txt"

theFeatures = []
theLabel = "CLASSIFICATION"
numFeatures = 0

data = loadCSV(dataFile)

for i in data.columns:
    if i != theLabel:
        theFeatures.append(i)

x = data.loc[:, theFeatures]
y = data[theLabel]

if t == 'TRAIN':

    #SCALE THE DATA
    scaler = RobustScaler()
    x = scaler.fit_transform(x)

    #TRAIN MODEL
    model = svm.LinearSVC()
    model.fit(x, y)

    #SAVE MODEL AND SCALER
    joblib.dump(scaler, scalerFile)
    joblib.dump(model, modelFile)

    rightNow = returnTimeFormatted()
    diff = secondsBetween(start, rightNow)

    with open(resultsFile, "a") as f:
        f.write("%s \r\n" %note)
        f.write('%s %s %s' % (e,t,m))
        f.write("*****\r\n")
        f.write("SCRIPT: %s\r\n" %pythonFileUsed)
        f.write("DATA FILE USED: %s\r\n" %dataFile)
        f.write("STARTED: %s\r\n" %start)
        f.write("FINISHED: %s\r\n" %rightNow)
        f.write("TIME ELAPSED IN SECONDS: %s\r\n" %diff)
        f.write("*****\r\n")
        f.write("MODEL USED: SVM\r\n")
        f.write("SETTINGS USED: %s\r\n" %model)
        f.write("*****\r\n")

    else:

        #SCALE THE DATA
        scaler = joblib.load(scalerFile)
        model = joblib.load(modelFile)

        x = scaler.transform(x)
        #TEST MODEL
        prediction = model.predict(x)

        rightNow = returnTimeFormatted()
        diff = secondsBetween(start, rightNow)

        cm = metrics.confusion_matrix(y, prediction)
        cr = metrics.classification_report(y, prediction)
        accuracy = metrics.accuracy_score(y, prediction)
        f1M = metrics.f1_score(y, prediction, average='macro')
        f1W = metrics.f1_score(y, prediction, average='weighted')
        precisionM = metrics.precision_score(y, prediction, average='macro')
        precisionW = metrics.precision_score(y, prediction, average='weighted')
        recallM = metrics.recall_score(y, prediction, average='macro')
        recallW = metrics.recall_score(y, prediction, average='weighted')

        #SVM / Metrics Print
        print("Confusion Matrix: \n", cm)
        print("Accuracy: ", round(accuracy,4))
        print("TIME ELAPSED IN SECONDS: %s\r\n" %diff)

```

```

with open(resultsFile,"a+") as f:
f.write("%s \r\n" %note)
f.write("*****\r\n")
f.write("SCRIPT: %s\r\n" %pythonFileUsed)
f.write("DATA FILE USED: %s\r\n" %dataFile)
f.write("STARTED: %s\r\n" %start)
f.write("*****\r\n")
f.write("MODEL USED: SVM\r\n")
f.write("SETTINGS USED: %s\r\n" %model)
f.write("*****\r\n")
f.write("Confusion Matrix: \r\n %s\r\n" %cm)
f.write("Classification Report: \r\n %s\r\n" %cr)
f.write("Accuracy: %s\r\n" %round(accuracy,4))
f.write("F1 Macro: %s\r\n" %round(f1M,4))
f.write("F1 Weighted: %s\r\n" %round(f1W,4))
f.write("Precision Macro: %s\r\n" %round(precisionM,4))
f.write("Precision Weighted: %s\r\n" %round(precisionW,4))
f.write("Recall Macro: %s\r\n" %round(recallM,4))
f.write("Recall Weighted: %s\r\n" %round(recallW,4))
f.write("Matthews Corr Coef: %s\r\n" %round(matthews_corrcoef,4))
f.write("*****\r\n")
f.write("*****\r\n")
f.write("*****\r\n")
f.write("FINISHED: %s\r\n" %rightNow)
f.write("TIME ELAPSED IN SECONDS: %s\r\n" %diff)
f.write("*****\r\n")

exit()

```

## Appendix I

### Genetic Feature Selection – Model B

```
#####
#AUTHOR: KATHY GOESCHEL
#DATE 09/02/2019
#PURPOSE: PHD DISSERTATION - NOVA SOUTHEASTERN UNIVERSITY
#ML MODEL FOR STATIC ANALYSIS CLASSIFICATION IMPROVEMENTS
#STATUS PRINTS AND RESULT EXPORTS REMOVED FROM SCRIPT PRIOR TO PUBLICATION
#####
#python Model_B.py

from datetime import datetime
import csv
import os
import sys
import time

import numpy as np
import pandas as pd
from random import random, randint

from sklearn.preprocessing import RobustScaler
from sklearn import svm

import sklearn.metrics as metrics

#####DECLARE PATHS / VARIABLES#####

pythonFileUsed = str(sys.argv[0]).split(".")[0]
experimentRun = str(sys.argv[1])
datetimeFormat = '%Y-%m-%d %H:%M:%S.%f'
scriptStart = datetime.now().strftime(datetimeFormat)

note = "MODEL B - SVM with GA - USING SAME TRAIN TEST DATA AS MODEL A -- MODEL: Basic Linear"

#####MAIN PATHS
mainDir = "/MODEL/"
trainData = ''
testData = ''

#####RESULTS FILE
#used to export results
resultsDir = mainDir + "RESULTS/"
if not os.path.exists(resultsDir + pythonFileUsed):
    os.mkdir(resultsDir + pythonFileUsed)
resultsDir = mainDir + "RESULTS/" + pythonFileUsed + "/"
resultsFile = resultsDir + pythonFileUsed + "_" + experimentRun + ".txt"
resultsFileCSV = resultsDir + pythonFileUsed + "_" + experimentRun + ".csv"

theLabel = "CLASSIFICATION"
numFeatures = 0
gensNoImprovements = 0

currGA = 0
currGABest = []
currGen = 0
prevGenPerf = 0
bestGenAcc = 0
kgGenImprov = []

gaSettings = [{'populationSize' : 50, 'generations' : 20, 'retain' : 0.8, 'randomKeep' : 0.05,
'mutationProbability' : .03, 'improvementThreshold' : .0003},
{'populationSize' : 50, 'generations' : 20, 'retain' : 0.75, 'randomKeep' : 0.03, 'mutationProbability'
: .02, 'improvementThreshold' : .003},
{'populationSize' : 50, 'generations' : 20, 'retain' : 0.70, 'randomKeep' : 0.01, 'mutationProbability'
: .025, 'improvementThreshold' : .003},
{'populationSize' : 100, 'generations' : 50, 'retain' : 0.8, 'randomKeep' : 0.05, 'mutationProbability'
: .03, 'improvementThreshold' : .0003},
{'populationSize' : 100, 'generations' : 50, 'retain' : 0.75, 'randomKeep' : 0.03,
'mutationProbability' : .02, 'improvementThreshold' : .003},
```



```

        {'populationSize' : 100, 'generations' : 50, 'retain' : 0.70, 'randomKeep' : 0.01,
'mutationProbability' : .025, 'improvementThreshold' : .003},
        {'populationSize' : 150, 'generations' : 50, 'retain' : 0.8, 'randomKeep' : 0.05, 'mutationProbability'
: .03, 'improvementThreshold' : .0003},
        {'populationSize' : 150, 'generations' : 50, 'retain' : 0.75, 'randomKeep' : 0.05,
'mutationProbability' : .03, 'improvementThreshold' : .0003},
        {'populationSize' : 150, 'generations' : 50, 'retain' : 0.70, 'randomKeep' : 0.05,
'mutationProbability' : .03, 'improvementThreshold' : .0003},
        {'populationSize' : 150, 'generations' : 100, 'retain' : 0.75, 'randomKeep' : 0.03,
'mutationProbability' : .02, 'improvementThreshold' : .003},
        {'populationSize' : 150, 'generations' : 100, 'retain' : 0.70, 'randomKeep' : 0.03,
'mutationProbability' : .02, 'improvementThreshold' : .003},
        {'populationSize' : 150, 'generations' : 100, 'retain' : 0.8, 'randomKeep' : 0.03,
'mutationProbability' : .02, 'improvementThreshold' : .003},
        {'populationSize' : 200, 'generations' : 500, 'retain' : 0.8, 'randomKeep' : 0.05,
'mutationProbability' : .03, 'improvementThreshold' : .0003},
        {'populationSize' : 200, 'generations' : 500, 'retain' : 0.75, 'randomKeep' : 0.03,
'mutationProbability' : .02, 'improvementThreshold' : .003},
        {'populationSize' : 200, 'generations' : 500, 'retain' : 0.70, 'randomKeep' : 0.01,
'mutationProbability' : .025, 'improvementThreshold' : .003},
    ]

#used to export results
gaResults = []
gaResultsKey = ()
#####FUNCTIONS#####

#####LOAD CSV
def loadCSV(file):
    temp = pd.read_csv(file, delimiter=',', skiprows=0).replace('","','')
    return temp

def returnTimeFormatted():
    return datetime.now().strftime(datetimeFormat)

def secondsBetween(d1, d2):
    d1 = datetime.strptime(d1, datetimeFormat)
    d2 = datetime.strptime(d2, datetimeFormat)
    return abs((d2 - d1).seconds)

def createSet(numFeatures):
    temp = bytearray()
    for i in range(numFeatures):
        temp.append(round(random()))
    return temp

def createPopulation(popSize, numFeatures):
    population = []
    for i in range(popSize):
        population.append(createSet(numFeatures))
    return population

def featureSetToNames(featureSet):
    temp = []
    x = 0
    for f in featureSet:
        if f == 1:
            temp.append(originalFeatures[x])
            x += 1
    return temp

def terminationCondition(val1, val2):
    if val1 >= val2:
        return True
    return False

def getFitness(p):
    tempSet = []

    for i in range(len(p)):
        if p[i] == 1:
            #this is a selected feature to be included in the feature set
            tempSet.append(originalFeatures[i])

    #MATCH NAMES FROM ORIGINAL SET TO NEW SET
    for j in newFeaturesMapping:

```

```

    for k in j:
        if k in tempSet:
            for x in j[k]:
                tempSet.append(x)
            tempSet.remove(k)
        return evaluateFeatureSet(tempSet)

def evaluateFeatureSet(tempSet):

    trainDataSubSet = trainData
    testDataSubSet = testData

    #remove features not selected
    for h in trainDataSubSet.columns:
        if h not in tempSet and h != theLabel:
            trainDataSubSet = trainDataSubSet.drop(columns=h)
            testDataSubSet = testDataSubSet.drop(columns=h)

    x_train = trainDataSubSet.loc[:, tempSet]
    y_train = trainDataSubSet[theLabel]

    x_test = testDataSubSet.loc[:, tempSet]
    y_test = testDataSubSet[theLabel]

    scaler = RobustScaler()
    x_train = scaler.fit_transform(x_train)
    x_test = scaler.transform(x_test)

    model = svm.LinearSVC()
    model.fit(x_train, y_train)
    prediction = model.predict(x_test)
    accuracy = metrics.accuracy_score(y_test, prediction)

    return accuracy

def evolve(population):
    global currGABest, currGA, currGen
    temp = []
    parents = []
    children = []
    y = 0

    #get the fitness of the population set
    for p in population:
        temp.append([y, getFitness(p)])
    y += 1

    #sort the population sets by increasing fitness
    temp = sorted(temp, key=lambda tup: tup[1], reverse=True)

    kgGenImprov.append((currGA, currGen, sum([c[1] for c in temp])/len(temp)))

    for x in temp:
        found = False
        for j in currGABest:
            if list(population[x[0]]) == j[3] and x[1] == j[2]:
                found = True
        if not found:
            ll = featureSetToNames(population[x[0]])
            currGABest.append([currGA, currGen, x[1], list(population[x[0]]), ll, len(ll)])
            currGABest = sorted(currGABest, key=lambda tup: tup[2], reverse=True)[:10]

    #save the top % and make them parents
    retainLength = int(len(temp)*retain)

    #keep the top performing sets as parents
    for i in range(retainLength):
        indexFromPopulation = temp[i][0]
        parents.append(population[indexFromPopulation])

    #include random poor performing sets for diversity
    for i in temp[retainLength:]:
        if randomKeep > random():
            indexFromPopulation = i[0]
            parents.append(population[indexFromPopulation])

```

```

#mutate a parent for additional diversity
for i in parents:
    if mutationProbability > random():
        positionToMutate = randint(0, len(i)-1)
        if i[positionToMutate] == 1:
            i[positionToMutate] = 0
        else:
            i[positionToMutate] = 1

#crossover parents to create children
while len(children) < (populationSize - len(parents)):
    male = randint(0, len(parents)-1)
    female = randint(0, len(parents)-1)
    if male != female:
        male = parents[male]
        female = parents[female]
    #CROSSOVER
    #split 50/50
    #half = int(len(male) / 2)
    #split randomly
    half = randint(0, numTotalFeatures-1)
    child = male[:half] + female[half:]
    children.append(child)

parents.extend(children)
return parents

#####START#####

#####LOAD / GET DATA SET
trainFile = mainDir + "DATA/MODEL_PRE-PROCESS/PreProcessed_" + experimentRun + "_TRAIN.csv"
trainData = loadCSV(trainFile)
testFile = mainDir + "DATA/MODEL_PRE-PROCESS/PreProcessed_" + experimentRun + "_TEST.csv"
testData = loadCSV(testFile)

#FILL IN THE ORIGINAL FEATURE LIST AS DICTLIST
originalFeatures = []

#GET THIS VALUE FROM THE PRE-PROCESSED TEXT FILE THAT PRINTS THE DELETED COLUMNS FROM PRE-PROCESSING STEP
(DICTLIST)
preProcessedDeletedFeatures = []

#GET THIS VALUE FROM THE PRE-PROCESSED TEXT FILE THAT PRINTS THE NEW COLUMN NAME MAPPINGS FROM PRE-
PROCESSING STEP (DICTLIST)
newFeaturesMapping = []

#ALL FEATURES ARE GATHERED FROM THE DATASET USED
allFeatures = []

for x in preProcessedDeletedFeatures:
    if x in originalFeatures:
        originalFeatures.remove(x)

allFeatures = []
for i in trainData.columns:
    if i != theLabel:
        allFeatures.append(i)

numTotalFeatures = len(originalFeatures)

#LOOP THROUGH ALL OF THE SPECIFIED GA SETTINGS AND RUN THOSE GAs ONE AFTER THE OTHER
for t in range(len(gaSettings)):

    currGA = t
    currGABest = []
    currGAWorst = []
    prevGenPerf = 0
    bestGenAcc = 0
    gensNoImprovements = 0

    for x in gaSettings[t]:
        if (x == 'populationSize') or (x == 'generations'):
            exec("%s = %d" % (x,gaSettings[t][x]))
        else:
            exec("%s = %.4f" % (x,gaSettings[t][x]))

```

```

maxGenerationsNoImprovements = int(generations / 10)

#CREATE INITIAL POPULATION
population = createPopulation(populationSize, numTotalFeatures)

termination = False
terminationGen = 0
topPerfomer = []
topPerformerNoChange = 0

#EVOLVE THE GENERATIONS
for i in range(0, generations):
    if terminationCondition(gensNoImprovements,maxGenerationsNoImprovements) or (topPerformerNoChange >
(maxGenerationsNoImprovements*1.5)):
        termination = True
        terminationGen = i
        break

    currGen = i

    kgStart = returnTimeFormatted()

    population = evolve(population)

    #NEW TOP PERFORMER
    if topPerfomer == currGABest[0]:
        topPerformerNoChange += 1
    else:
        topPerformerNoChange = 0
        topPerfomer = currGABest[0]

    thisGenPerf = (sum([c[2] for c in currGABest])/len(currGABest))

    #detect the accuracy delta to be sufficient between generations or multiple gens
    if ((thisGenPerf - prevGenPerf) > improvementThreshold):
        gensNoImprovements = 0
    else:
        gensNoImprovements += 1

    if thisGenPerf > bestGenAcc:
        bestGenAcc = thisGenPerf
        gensNoImprovements = 0

    prevGenPerf = thisGenPerf

    gaResults.append(tempRes)

exit()

```

## References

- Abunadi, I., & Alenezi, M. (2015, September). Towards cross project vulnerability prediction in open source web applications. *In Proceedings of the The International Conference on Engineering & MIS 2015* (p. 42). ACM.
- Ambusaidi, M., He, X., Nanda, P., & Tan, Z. (2016). Building an intrusion detection system using a filter-based feature selection algorithm. *IEEE Transactions on Computers*, 65(10), 2986–2998.
- Ayewah, N., & Pugh, W. (2010, July). The google findbugs fixit. *In Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10* (pp. 241–252). ACM.
- Badshah, N. (2018). *Facebook to contact 87 million users affected by data breach*. Retrieved from <https://www.theguardian.com/technology/2018/apr/08/facebook-to-contact-the-87-million-users-affected-by-data-breach>
- Bell, R. M., Ostrand, T. J., & Weyuker, E. J. (2006, July). Looking for bugs in all the right places. *In Proceedings of the 2006 international symposium on Software testing and analysis* (pp. 61–72). ACM.
- Beller, M., Bholanath, R., McIntosh, S., & Zaidman, A. (2016, March). Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. *In 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (saner)* (pp. 470–481). IEEE.
- Bishop, P., Gashi, I., Littlewood, B., & Wright, D. (2007, November). Reliability modeling of a 1-out-of-2 system: Research with diverse off-the-shelf sql database servers. *In Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on* (pp. 49–58). IEEE.
- Bleier, J. (2017). *Improving the Usefulness of Alerts Generated by Automated Static Analysis Tools (Unpublished doctoral dissertation)*. Radboud University Nijmegen.
- Buczak, A., & Guven, E. (2015). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2), 1153–1176.
- Carrozza, G., Cinque, M., Giordano, U., Pietrantuono, R., & Russo, S. (2015, May). Prioritizing correction of static analysis infringements for cost-effective code sanitization. *In Proceedings of the Second International Workshop on Software Engineering Research and Industrial Practice* (pp. 25–31). IEEE Press.
- Chen, H., & Wagner, D. (2002, November). MOPS: an Infrastructure for Examining Security Properties of Software. *In Proceedings of the 9th acm conference on computer and communications security* (pp. 235–244). ACM.
- Chess, B., & McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, 2(6), 76–79.
- Chimdylwar, B., & Kumar, S. (2011, February). Effective false positive filtering for evolving software. *In Proceedings of the 4th India Software Engineering Conference* (pp. 103–106). ACM.
- Cisco. (2017). *Annual Cyber Security Report* (Tech. Rep.). Retrieved from <http://b2me.cisco.com/en-us-annual-cybersecurity-report-2017>
- Delaitre, A., Stivalet, B., Fong, E., & Okun, V. (2015, May). Evaluating Bug Finders. In *Complex faults and failures. In large software systems (coufless), 2015 IEEE/ACM 1st international workshop on* (pp. 14–20). IEEE.

- Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 42–51.
- FBI. (2017). *Internet crime report* (Tech. Rep.). Federal Bureau of Investigation.
- Fry, Z. P., & Weimer, W. (2013, October). Clustering static analysis defect reports to reduce maintenance costs. *In Reverse Engineering (WCRE), 2013 20th Working Conference on* (pp. 282–291). IEEE.
- Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68, 18–33.
- Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting Fault Incidence Using Software Change History. *IEEE Transactions on software engineering*, 26(7), 653–661.
- Hanam, Q., Tan, L., Holmes, R., & Lam, P. (2014, May). Finding patterns in static analysis alerts: improving actionable alert ranking. *In Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 152–161). ACM.
- Heaton, J. B., Polson, N. G., & Witte, J. H. (2017). Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1), 3–12.
- Heckman, S., & Williams, L. (2008, October). On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. *In Proceedings of the second acm-ieee international symposium on empirical software engineering and measurement* (pp. 41–50). ACM.
- Heckman, S., & Williams, L. (2009, April). A model building process for identifying actionable static analysis alerts. *In Software Testing Verification and Validation, 2009. ICST'09. International Conference on* (pp. 161–170). IEEE.
- Heckman, S., & Williams, L. (2013, October). A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques. *In Proceedings of the 9th international conference on predictive models in software engineering* (p. 4). ACM.
- Heckman, S. S. (2007). Adaptively ranking alerts generated from automated static analysis. *Crossroads*, 14(1), 7.
- Herter, J., Daniel, K., Mallon, C., Wilhelm, R., & Gmbh, A. (2017, September). Benchmarking Static Code Analyzers. *In International conference on computer safety, reliability, and security* (pp. 197–212). Springer, Cham.
- Holland, J. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press.
- Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *Acm sigplan notices*, 39(12), 92–106.
- Hovsepian, A., Scandariato, R., & Joosen, W. (2016, September). Is Newer Always Better?: The Case of Vulnerability Prediction Models. *In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (p. 26). ACM.
- Hussain, L., Aziz, W., Saeed, S., Rathore, S., & Rafique, M. (2018, August). Automated Breast Cancer Detection Using Machine Learning Techniques by Extracting Different Feature Extracting Strategies. *In 2018 17th ieee international conference on trust, security and privacy in computing and communications/ 12th ieee international conference on big data science and engineering (trustcom/bigdatase)* (pp. 327–331). IEEE.
- Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). Why don't software developers use static analysis tools to find bugs? *In Proceedings of the 2013 International Conference on Software Engineering* (pp. 672–681). IEEE.

- Johnson, S. C. (1978). Lint, a C Program Checker. *Comp. Sci. Tech. Rep.*, 78–1273.
- Kim, S., & Ernst, M. D. (2007a, May). Prioritizing warning categories by analyzing software history. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on* (pp. 27–27). IEEE.
- Kim, S., & Ernst, M. D. (2007b, September). Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering - ESEC-FSE '07* (pp. 45–54). ACM.
- Koc, U., Saadatpanah, P., Foster, J. S., & Porter, A. A. (2017, June). Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st acm sigplan international workshop on machine learning and programming languages - mapl 2017* (pp. 35–42). ACM.
- Kourou, K., Exarchos, T. P., Exarchos, K. P., Karamouzis, M. V., & Fotiadis, D. I. (2015). Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 13, 8–17.
- Kowalczyk, A. (2017). *Support Vector Machines Succinctly*. Syncfusion, Inc.
- Kremenek, T., Ashcraft, K., Yang, J., & Engler, D. (2004). Correlation exploitation in error ranking. *ACM SIGSOFT Software Engineering Notes*, 29(6), 83–93.
- Kremenek, T., & Engler, D. (2003, June). Z-ranking: Using statistical analysis to counter. In *International Static Analysis Symposium* (pp. 295–315). Springer, Berlin, Heidelberg.
- Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R. P., Tang, J., & Liu, H. (2018). Feature Selection: A Data Perspective. *ACM Computing Surveys (CSUR)*, 50(6), 94.
- Medeiros, I., Neves, N., & Correia, M. (2016). Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1), 54–69.
- Medeiros, I., Neves, N. F., & Correia, M. (2014, April). Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives. In *Proceedings of the 23rd international conference on world wide web* (pp. 63–74). ACM.
- Munson, J. C., & Khoshgoftaar, T. M. (1992). The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5), 423–433.
- Muske, T., & Serebrenik, A. (2016, October). Survey of approaches for handling static analysis alarms. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (pp. 157–166). IEEE.
- NIST. (2017a). *JULIET Test Suite*. Retrieved from <https://samate.nist.gov/SARD/testsuite.php>
- NIST. (2017b). *NIST*. Retrieved from <https://www.nist.gov/>
- Nunes, P., Medeiros, I., Fonseca, J., Neves, N., Correia, M., & Vieira, M. (2017, September). On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *2017 13th European Dependable Computing Conference (EDCC)* (pp. 121–128). IEEE.
- Ogasawara, H., Aizawa, M., & Yamada, A. (1998, November). Experiences with program static analysis. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International* (pp. 109–112). IEEE.
- Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004, July). Where the bugs are. In *Acm sigsoft software engineering notes* (Vol. 29, pp. 86–96). ACM.
- Pang, Y., Xue, X., & Wang, H. (2017, June). Predicting Vulnerable Software Components through Deep Neural Network. In *Proceedings of the 2017 International Conference on Deep Learning Technologies - ICDLT '17* (pp. 6–10). ACM.
- Podelski, A., Schäfer, M., & Wies, T. (2016, July). Classifying bugs with interpolants. In *International Conference on Tests and Proofs* (pp. 151–168). Springer, Cham.
- Python Software Foundation. (n.d.). *Python*. Retrieved from <https://www.python.org/>
- R Core Team. (2013). *R: A language and environment for statistical computing*. Vienna, Austria. Retrieved from <http://www.R-project.org/>
- Reynolds, Z. P., Jayanth, A. B., Koc, U., Porter, A. A., Raje, R. R., & Hill, J. H. (2017, May). Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. In *Proceedings - 2017 ieee/acm 4th international workshop on software engineering research and industrial practice, ser and ip 2017* (pp. 55–61). IEEE.
- Russell, S., & Norvig, P. (2014). *Artificial Intelligence A Modern Approach*. New Jersey: Pearson.
- Ruthruff, J. R., Penix, J., Morgenthaler, J. D., Elbaum, S., & Rothermel, G. (2008, May). Predicting accurate and actionable static analysis warnings. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 341–350). ACM.
- Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10), 993–1006.

- Shin, Y., & Williams, L. (2013). Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1), 25–59.
- Shiraishi, S., Mohan, V., & Marimuthu, H. (2015, November). Test Suites for Benchmarks of Static Analysis Tools. *In Software reliability engineering workshops (issrew), 2015 ieee international symposium on* (pp. 12–15). IEEE.
- Shivaji, S., Whitehead, E. J. J., Akella, R., & Kim, S. (2013). Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4), 552–569.
- Sun, T., & Vasarhelyi, M. A. (2018). Predicting credit card delinquencies: An application of deep neural networks. *Intelligent Systems In Accounting, Finance and Management*, 25(4), 174–189.
- Tripp, O., Pistoia, M., & Aravkin, A. (2014, November). A LETHEIA: Improving the Usability of Static Security Analysis. *In Proceedings of the 2014 acm sigsac conference on computer and communications security* (pp. 762–774). ACM.
- Understand. (n.d.). *SciTools*. Retrieved from <https://scitools.com/>
- US Senator Elizabeth Warren. (2018). *Bad Credit: Uncovering Equifax's Failure to Protect American's Personal Information* (Tech. Rep. No. February). United States Senate.
- Verizon. (2018). 2018 Data Breach Investigations Report (DBIR). *Verizon Business Journal*.
- Walden, J., Stuckman, J., & Scandariato, R. (2014). Predicting vulnerable components: Software metrics vs text mining. *In Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on* (pp. 23–33). IEEE.
- Wang, Q., Meng, N., Zhou, Z., Li, J., & Mei, H. (2008, December). Towards SOA-based Code Defect Analysis. *In Service-oriented system engineering, 2008. sose'08. ieee international symposium on* (pp. 269–274). IEEE.
- Wedyan, F., Alrmuny, D., & Bieman, J. M. (2009, April). The effectiveness of automated static analysis tools for fault detection and refactoring prediction. *In Software Testing Verification and Validation, 2009. ICST'09. International Conference on* (pp. 141–150). IEEE.
- Williams, C. C., & Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6), 466–480.
- Xue, B., Zhang, M., Browne, W. N., & Yao, X. (2016). A Survey on Evolutionary Computation Approaches to Feature Selection. *IEEE Transactions on Evolutionary Computation*, 20(4), 606–626.
- Yan, M., Zhang, X., Xu, L., Hu, H., Sun, S., & Xia, X. (2017, July). Revisiting the Correlation Between Alerts and Software Defects: A Case Study on MyFaces, Camel, and CXF. *In Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual* (Vol. 1, pp. 103–108). IEEE.
- Yi, K., Choi, H., Kim, J., & Kim, Y. (2007). An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 102(2-3), 118–123.
- Yoon, J., Jin, M., & Jung, Y. (2014, December). Reducing false alarms from an industrial-strength static analyzer by SVM. *In Proceedings - asia-pacific software engineering conference, apsec* (Vol. 2, pp. 3–6). IEEE.
- Yüksel, U., & Sözer, H. (2013, September). Automated classification of static code analysis alerts: A case study. *In Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (pp. 532–535). IEEE.
- Zhang, D., Jin, D., Xing, Y., Zhang, H., & Gong, Y. (2013, July). Automatically mining similar warnings and warning combinations. *In Fuzzy Systems and Knowledge Discovery (FSKD), 2013 10th International Conference on* (pp. 783–788). IEEE.
- Zhang, Y., Lo, D., Xia, X., Xu, B., Sun, J., & Li, S. (2016, January). Combining Software Metrics and Text Features for Vulnerable File Prediction. *In Proceedings of the ieee international conference on engineering of complex computer systems, iceccs* (pp. 40–49). IEEE.
- Zhioua, Z., Short, S., & Roudier, Y. (2014a, July). Static code analysis for software security verification: Problems and approaches. *In Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International* (pp. 102–109). IEEE.
- Zhioua, Z., Short, S., & Roudier, Y. (2014b). Towards the verification and validation of software security properties using static code analysis. *International Journal of Computer Science: Theory and Application*, 2.