

2015

# Static Code Analysis: On Detection of Security Vulnerabilities and Classification of Warning Messages

Andrei M. Perhinschi

# Static Code Analysis: On Detection of Security Vulnerabilities and Classification of Warning Messages

by

Andrei M. Perhinschi

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

Master of Science
in
Computer Science

Katerina Goseva-Popstojanova, Ph.D., Chair
Hany Ammar, Ph.D.
Roy Nutter, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2015

Keywords: static code analysis, machine learning, information assurance

## Abstract

This thesis addresses several aspects of using static code analysis tools for detection of security vulnerabilities and faults within source code. First, the performance of three widely used static code analysis tools with respect to detection of security vulnerabilities is evaluated. This is done with the help of a large benchmarking suite designed to test static code analysis tools' performance regarding security vulnerabilities. The performance of the three tools is also evaluated using three open source software projects with known security vulnerabilities. The main results of the first part of this thesis showed that the three evaluated tools do not have significantly different performance in detecting security vulnerabilities. 27% of C/C++ vulnerabilities along with 11% of Java vulnerabilities were not detected by any of the three tools. Furthermore, overall recall values for all three tools were close to or below 50% indicating performance comparable or worse than random guessing. These results were corroborated by the tools' performance on the three real software projects. The second part of this thesis is focused on machine-learning based classification of messages extracted from static code analysis reports. This work is based on data from five real NASA software projects. A classifier is trained on increasing percentages of labeled data in order to emulate an on-going analysis effort for each of the five datasets. Results showed that classification performance is highly dependent on the distribution of true and false positives among source code files. One of the five datasets yielded good predictive classification regarding true positives. One more dataset led to acceptable performance, while the remaining three datasets failed to yield good results. Investigating the distribution of true and false positives revealed that messages were classified successfully when either only real faults and/or only false faults were clustered in files or were flagged by the same checker. The high percentages of false positive singletons (files or checkers that produced 0 true positives and 1 false negative) were found to negatively affect the classifier's performance.

# Acknowledgments

The work presented in this thesis was funded in part by the NASA Independent Verification and Validation Facility in Fairmont, WV through grants managed by TASC Inc. The author thanks Brandon Bailey, Keenan Bowens, Richard Brockway, Van Casdorph, Travis Dawson, Roger Harris, Vaughn Harvey, Joelle Loretta, Shirley Savarino, Jerry Sims and Christopher Williams for their input and feedback.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Static analysis of source code provides a scalable method for code review and helps ensure that coding policies are being followed. Tools for static analysis have rapidly matured in the last decade; they have evolved from simple lexical analysis to employ much more accurate and complex techniques. Many commercial and open source tools exist, and all of them make certain claims about their effectiveness and performance.

However, in general, static analysis problems are undecidable [8] (i.e., it is impossible to construct an algorithm which always leads to a correct answer in every case). Therefore, static analysis tools do not detect all faults in source code (false negatives) and are prone to false positives (i.e., they report findings which upon closer examination turn out not to be faults at all). Therefore to be of practical use, a static code analysis tool should find as many faults as possible, ideally all, with a minimum amount of false positives, ideally none.

## 1.1 Introduction to Static Code Analysis Tool Evaluation

The advances in technology and broadband connectivity, combined with ever increasing number of threats and attacks, require information assurance and cyber security to be integrated into the traditional verification and validation process. Many organizations develop, run, and maintain numerous systems for which one or more security attributes (i.e., confidentiality, integrity, availability, authentication, authorization, and non-repudiation) are of vital importance. Therefore it is becoming imperative to extend current validation and verification capabilities to cover information

assurance and cyber security concerns for sensitive projects.

Vulnerabilities are more costly to fix than prevent in the first place [6]. To be able to prevent vulnerabilities from ever existing, the development process needs to be tailored towards this goal. Related work shows that static code analysis is only one part of a successful development process [3], [34]. What is unclear is whether or not static analysis tools deliver what they promise. Therefore the motivation for this work was to determine whether static code analysis tools display significant differences in detection of security vulnerabilities and if so where these differences occur. In this thesis the capabilities of three existing state-of-the-practice static code analysis tools were explored. Previous efforts include the work done at the multiple SATE events [29]. While examining more tools than the work presented here, the SATE approach did not include any rigorous statistical testing. Instead it relied only on the reporting of metrics such as recall and precision [30]. Similar to the work done within the SATE conference, the experiment presented in this thesis makes use of the Common Weakness Enumeration (CWE) nomenclature, which is a hierarchical classification of security vulnerability types. A given CWE will either denote one particular type of security vulnerability, for example buffer overflow or SQL injection, or it will represent a vulnerability class, such as numeric errors or coding standard violations. CWEs are described in more detail in Section 3.1.

The main motivation behind this work can be summarized with the following three research questions:

- R1: Is there a difference between three state of the practice static code analysis tools with respect to detection of security related faults when applied to a benchmark suite?
- R2: If so which of the tools performs the best?
- R3: Is the performance of the three static code analysis tools on real software applications consistent with the performance on the benchmark suite?

The three tools evaluated in this thesis were chosen based on factors such as popularity, ease of use, and the ability to check partial source code. This work is based on a large benchmark suite [23] consisting of 22 vulnerability ”classes” for C/C++ and 19 for Java, with approximately 20,000 and 7,500 test cases respectively. The benchmark-based evaluation was conducted auto-

matically by the use of a scripting system developed as part of this effort. The methodology is based on automatic extraction of quantitative metrics from the outputs of the three static analysis tools, denoted throughout this thesis as Tools A, B, and C. The work presented here is an empirical study consisting of a controlled experiment with benchmark input. The Juliet test suite was run through each of the three tools after which the tools' output was taken and ran through a script system developed to compute detection performance metrics. The results were empirically evaluated by employing statistical hypothesis testing in order to determine any significant differences between the three tools' performance. The inclusion of rigorous statistical testing is an important contribution as compared to the related work.

In addition to the automatic benchmark-based experiment, the evaluation was extended with three open source programs, two of which were implemented in C (with 8,500 and 280,000 LOC, respectively) and one implemented in Java (with 4,800,000 LOC). All three of the programs selected have known sets of vulnerabilities in order to allow quantitative analysis. For this part of the study all results are obtained by manual inspection of the static tools' outputs as there was no way of automating the process, nor was it necessary due to the relatively small number of known defects.

The contribution of this work can be shortly summarized as follows:

- Approach
  - The experimental approach employed in this work was based on a benchmark suite called Juliet developed by the NSA Center for Assured Software. The main purpose of the Juliet suite is to assess static code analysis tools. Earlier studies based on the Juliet suite either employed a very small subset of the C test cases [12] or did not provide quantitative results [7] [26].
  - The work presented here provides a number of quantitative performance metrics: accuracy, recall, probability of fasle alarm, balance, and G-score. These metrics are reported per individual CWE and also across all CWEs. Formal statistical testing, in the form of the Friedman rank sum test, was employed in comparing the tools' performance metrics in order to determine the existence of any significant difference in performance. This is a considerable improvement over the referenced related works, none of which

reported any such statistical tests.

  - Three popular open source applications were employed as case studies besides the experimental approach. These were included as a way of verifying the experimental results based on the Juliet benchmark in a more realistic situation.

- Empirical observations

  - The three tools were all unable to detect all vulnerabilities. In this context the term "detect" refers to correctly classifying at least one bad function; it does not mean being able to detect each and every instance of faulty code for a given CWE. None of the tools were able to detect six of the 22 C/C++ CWEs. Furthermore seven C/C++ CWEs were detected either by a single tool or a combination of two tools. All three tools were able to detect only nine of C/C++ CWEs. The evaluation of Java CWEs yielded similar results. None of the tools detected two of the 19 Java CWEs while thirteen Java CWEs were detected either by a single tool or a combination of two tools. All three tools were able to detect only four of the Java CWEs.
  - The three static analysis tools showed no statistically significant difference in detecting security vulnerabilities. This lack of significant difference was observed for both C/C++ and Java test cases. Furthermore, the mean, median, weighted, and overall values for the recall metric were either about or less than 50%. This signifies similar or worse performance than random chance.
  - One of the tools displayed better performance with respect to probability of false alarm for the C/C++ test cases. However, no such difference was measured for Java test cases.
  - The three open source program case studies confirmed the experimental results with respect to the tools' security vulnerability detection performance.

## 1.2 Introduction to Static Code Analysis Message Classification Experiments

Being an integral part of the software development process, static code analysis has the potential to improve the quality of software. Static code analysis consists of running source code through a software tool which then outputs a list of messages. Static analysis tools employ pattern matching constructs known as ”checkers” which flag pieces of source code as defective if the source code happens to match a particular checker. For example, a static analysis tool has a checker designed to detect possible buffer overflow conditions. When the tool encounters a piece of source code which matches its buffer overflow checker, a message is produced. Each message (often called a warning) consists of information related to a specific piece of source code that the static analysis tool has flagged as faulty. The information provided varies among static analysis tools; however a majority of tools report the following: source code filename and function or method where potentially faulty code is located, line number(s) where faulty code is located, some type of severity ranking of the fault, and checker which flagged the fault.

Throughout the rest of this thesis the list of messages output by static analysis tools is referred to as ”report”. Each report contains many messages, up to hundreds of thousands for large projects with millions of lines of code. Each message is either a true positive (TP) or a false positive (FP). True positives are actual code faults while false positives are not. For each message this determination falls to a software analyst who has to manually inspect the code referenced by the message and decide whether or not the static analysis software has made a mistake. In practice, static analysis tools produce large numbers of false positives [14], [4] which makes it difficult for analysts and developers to put such tools to effective use. It is clear that time expended identifying false positives would be better spent on other tasks, such as fixing true positives.

Analysts’ competence at dealing with the large numbers of false positives resulting from static code analysis was studied in [5]. The findings showed that, while analysts were quite competent with respect to true positive classification, they were not very successful in identifying false positives. Furthermore, it was reported that analyst experience did not make a difference with regard to their ability to identify false positive messages. A newer study by the same research team [4] again confirmed the negative effects of false positives. The results showed a high incidence of false

positives among static analysis tool output, with slightly more than a third of them being identified as such by analysts. The researchers also found that false positives negatively impacted analysts' efforts by forcing them to spend time looking at and identifying non-issues. These results further confirm that some classification of true and false positives should be an integral part of any static analysis effort. It follows that if the number of false positives could be significantly decreased, the static code analysis process could be made less time consuming and would lead to better outcomes in terms of software quality, including reliability and security.

The motivation for this work was to evaluate the viability of machine learning in classifying static code analysis report messages using only features extracted from the static code analysis report. While integrating data from more than just the static analysis tool may make it more likely for machine learning algorithms to perform well, such integration between static analysis tool reports and other sources is not always possible. It would then be desirable to be able to perform machine learning classification using only static analysis reports. Specifically, the following research questions are posed:

- R4: Can static analysis messages be successfully classified to true positives and false positives?
- R5: If so, what percentage of report messages must be labeled before predictions become useful?
- R6: What is the distribution of true and false positives among files and checkers?

Exploring the distribution of true and false positives across files was motivated by the so-called Pareto principle, that is that around 80% of faults are localized in 20% of files. The Pareto principle has been confirmed in numerous studies [27], [1], [18], [11], with the actual percentage of faults contained in about 20% of files ranging from 60% to 90%.

Furthermore, it was found that many checkers also tend to cluster false positives for various reasons such as the inherent vagueness involved when designing such checkers and the variety of possible code constructs that may violate checkers' rules about what constitutes a correct construct while not actually being incorrect [24]. This motivated the decision to look at true and false positive distribution across checkers.

The classification approach used in this work is based on using the Classification Based on Associations (CBA) algorithm [25]. CBA is an association rule-based classifier which works on categorical data such as the fields found in static code analysis reports. The experimental design is based on train/test experimentation with incremental percentages of a given dataset being trained on. Training was done on 25% and testing on 75% of a given dataset, then training was done on 50% and testing on 50%, then training on 75% and testing on 25%, and finally training on 90% and testing on 10%. This setup was chosen to simulate an ongoing analysis process at different points in time. For the experiments presented in this part of the thesis we used static analysis reports from five real NASA projects which were manually analyzed, that is, the messages were manually checked by a software analyst.

The contributions of this part of the thesis are:

- The experiment presented in this thesis employed association rule based classification with training on different percentages of labeled messages.
- The work presented here applies classification on five datasets obtained from real NASA software projects' static code analysis efforts.

The main empirical observation of this work are as follows:

- The experiment presented in this thesis showed that two of the projects produced good classification results while the remaining three projects did not lend themselves well to association based classification.
- Classification based on association rules led to good performance when TP and FP messages were clustered in code units (files in the present case) with other messages of their type (TP or FP, respectively). When operating on datasets with code units containing mixed distributions of TP and FP messages performance was significantly decreased.
- Classification based on association rules did not lead to good performance on datasets with large numbers of FP singletons, that is when files produce only one message, which provide no benefit when training the classifier.

- Unlike the Stanford team [24] which found that singleton faults tend to appear quite balanced, that is TP and FP singletons seem to occur in relatively similar numbers, the results of this work show a significant trend towards FP singletons. More importantly, it was found that for some projects true and false positives tend to be found mixed together within files or checkers.

# Chapter 2

# Related Work

This chapter presents related work to both parts of this thesis, the static code analysis tool evaluation and the static code analysis message classification. The related works are split into two sections, respectively, for each main part of this thesis.

## 2.1 Related Work for Static Code Analysis Tool Evaluation

Despite the prevalent use of static analysis, few evaluation efforts of static code analysis tools based on repeatable benchmarking systems have been undertaken, and even fewer with a focus on security vulnerability detection. The following related works are based on static code analysis with some kind of benchmarking input.

The SAMATE (Software Assurance Metrics and Tool Evaluation) project, sponsored by the U.S. Department of Homeland Security (DHS) National Cyber Security Division and NIST, hosts the Static Analysis Tool Exposition [29], which annually reviews a wide variety of different tools. A recent study by the NSA [15], presented at the fourth SATE conference, tested eight commercial tools and one open source static analysis tool for C/C++ and seven commercial and one open source tool for Java. In that study, the evaluated tools were not identified and specific results for each tool were not made public. However, it was reported that 21% of the vulnerabilities in the C/C++ test cases and 27% of vulnerabilities in Java test cases were not detected by any tool. Furthermore, although open source tools detected some vulnerabilities, they were not among the best in any class of test cases.

A similar study, based on creating a test suite consisting of injected vulnerabilities in otherwise clean code, was conducted by the University of Hamburg in collaboration with the Siemens CERT [22]. Like the NSA evaluation, this work examined the performance of static code analysis tools in detecting security vulnerabilities. The study presented in [22] compared the performance of six tools. The findings showed that tools for analyzing C code scored within a range of 50-72 points out of the possible 168, and that tools for analyzing Java code scored within a range of 53-89 points out of the possible 147. Similar to the work done by NSA, this study did not identify the evaluated tools or the individual results.

A 2014 study [32] evaluated 2 commercial static analysis tools against the Juliet test suite. The Velicheti study was focused on evaluating tool performance with respect to code complexity. The authors found that both tools performed better at code with low cyclomatic complexity rather than high. Furthermore the researchers concluded that static analysis results are highly variable depending on the tool used, the type of vulnerability in question, and even the specific code construct in question.

A study [21] presented a comparison of 12 static code analysis tools, 11 open source and one commercial tool. This study was focused only on qualitative comparison using metrics such as installation process, configuration, support, reports understand, vulnerabilities coverage degree and support for handling projects. Although interesting, the study lacked quantitative comparison of tools performance.

The comparison of three commercials tools, Polyspace, Prevent and Klocwork Insight published in [13] concluded that Prevent and Klocwork found largely disjoint sets of vulnerabilities. The study also indicated that Prevent and Klocwork Insight sacrificed finding vulnerabilities in favor of reducing the number of false positives, while PolySpace produced a high rate of false positives.

A recent study [12] compared the performance of nine tools, most of them commercial, with respect to detection of security vulnerabilities in C code. The evaluation results presented in [12] were based on a very small benchmark test suite consisting of only 73 test cases for a total of only 22 CWEs. The examined tools had mean recall values in the range from 37.6% to 66.5%. Seven out of 22 CWEs were not detected by any tool. The results in [12] were consistent with those from [13] regarding the lack of detection overlap between Prevent and Klocwork. In agreement with

previous work, the study recommended that static analysis be integrated into the defect detection and removal process alongside other methods. Furthermore the study concluded that current generation static analysis tools cannot be employed interchangeably as their internal designs, as well as output formats, yield greatly differing results.

While many related studies conclude that false positives are a major issue in the application of static analysis, the effects of high numbers of false positives on the development process are rarely discussed. Understanding the implications of false positives is paramount when selecting a static code analysis tool to be integrated into a development environment. The following related works either looked at static code analysis tools from an environment integration point of view or they compare static analysis tools to other approaches such as penetration testing as part of an integrated development environment.

A recent study [4] investigated how static analysis should be integrated into the development process in order to assure successful usage and improvement in software quality with regards to security. The study focused on 4 commercial products from Ericsson analyzed with a single static analysis tool. The researchers concluded that most of the tool output was not security related and that false positives were an issue, albeit not as much as other studies with a false positive rate ranging from 5% to 20% depending on the product. More interestingly the study showed that only 37.5% of false positives were correctly identified as false positives. Furthermore it was found that developers may introduce new vulnerabilities while attempting to fix code that did not need to be fixed (that is the remaining 62.5% of false positives which are not identified as such). While this may be explained by a lack of developer experience, the same research team found that experience has no effect on developers false positive classification accuracy [5]. The study [5] claimed that while the developers used in the study were good at identifying true positives, false positive identification rates were not better than chance. Additionally developers were found to be bad at judging the accuracy of their own classifications. These conclusions indicated that high numbers of false positives were detrimental to the development process not only through the amount of time spent classifying them, but also through the added risk of fixing improperly classified false positives.

Another study regarding static code analysis with respect to security was aimed at web service vulnerabilities [2]. Specifically the study attempted to compare penetration testing and static

code analysis with regard to SQL injection vulnerabilities. The researchers chose three commercial automated penetration testing tools along with one testing tool proposed in previous work. These penetration testing tools were compared with three static code analysis tools by employing all tools over a set of eight web services. The researchers key metrics of interest were simply vulnerability coverage and false positive rate. This study concluded with three key observations: static analysis provided more coverage than penetration testing, false positives were a problem for both approaches but static analysis was affected more, and different tools often reported different vulnerabilities for the same piece of source code.

Taking an economic standpoint, yet another study evaluated the effectiveness of static analysis as pertaining to large organizations efforts in improving software quality [34]. The researchers focused on static analysis with Flexelint, Klocwork, and Illuma as applied to 3 large network service products from Nortel. In comparing the three tools defect removal rates with manual inspection the study found no significant difference. Furthermore the defect removal rates of execution-based testing were found to be two to three times higher than that of static analysis techniques. While this study was not exclusively focused on security vulnerabilities and these only comprised a small subset of the vulnerabilities explored, the study concluded that while static analysis can be used to identify certain programming errors with the potential to manifest as security vulnerabilities, static analysis is essentially complementary to other defect detection methods.

A 2013 study [3] compared 4 vulnerability detection techniques: exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis. The study used 3 large software projects. The authors found that static analysis resulted in many more false positives than the other 3 methods. They also found that static analysis did quite poorly on certain vulnerability types.

The work presented in this thesis takes a quantitative approach based on a standard software vulnerability nomenclature, namely the Common Weakness Enumeration [9]. Furthermore the way the CWE hierarchy is employed is indirectly through the automatically generated Juliet test suite which provides a repeatable benchmark. Despite the shortcomings of the CWE nomenclature itself, this approach is considered to be more straightforward and objective than methods based on researchers own categorizing of vulnerabilities, which much of the related work employs. In addition to using a repeatable benchmark statistical hypothesis testing was employed in the form

of the Friedman rank sum test to draw empirical conclusions about the results.

## 2.2 Related Work for Static Code Analysis Message Classification Experiments

The most primitive technique of dealing with large numbers of expected false positives is simple statistical sampling of reports followed by exhaustive manual analysis of the sample. The obvious issue with this simple method is the probability that significant true positives may not be part of the sample and be missed by analysts; in other words the sample of error reports might not be representative of the overall distribution. The fact that automated methods for message classification can cover the entirety of reports makes them attractive alternatives. One such method is based on error ranking schemes which apply a ranking centered on some metrics to static analysis reports in an attempt to push the most likely true positives to the front of the list and the most likely false positives to the bottom [24]. The researchers introduced a Bayesian incremental message ranking technique which takes into account newly inspected messages. They examined the clustering behavior of true and false positives within the source code of two large software systems. The researchers observed that when graphing the number of false vs. true positives per source code unit (file and function) the data fell into a particular "L" shape. The "sides" of the "L" shaped graph represent the distribution of source code units which contain only true positives and those which contain only false positives. These source code units which cluster along the axes of the TP/FP per source code unit charts represent the most predictively powerful data from a classification point of view. The source code units which cluster further away from the axes contain a mix of true and false positives. If these observations hold for other projects then it may be possible to apply machine learning techniques in order to perform classification on static analysis reports.

Many automated methods for message classification described in literature are based on some kind of machine learning. The end goal of such methods is to predict whether or not a previously unseen message will be a true or false positive based on a previously learned model, a very attractive prospect for the beleaguered analyst. Machine learning methods described in literature tend to integrate static code analysis report information with data from other sources such as static code

metrics, churn metrics, complexity metrics, etc, [28], [20].

A 2008 study [28] introduced four classification models for true and false positives based on features extracted from static analysis reports, such as checker, category, and priority. In addition to the features extracted from static analysis reports, the researchers also included other features, such as number of messages per file and several different churn metrics. The four models were evaluated with static analysis reports obtained by running Googles Java codebase through the open source FindBugs [16] static analysis tool. The study only reported precision metrics: 61-72% for true positive classification and 77-88% for false positive classification.

A code study [20] presented a comparison of fifteen machine learning algorithms applied to static analysis reports. The algorithms were evaluated using two real JAVA software projects. Similar to Ruthruff et al., this study used a mixture of features obtained from several sources. In addition to the features extracted directly from static analysis reports, the researchers also included source code features, such as cyclomatic complexity, code churn features, as well as other features such as file age and number of open issues per revision. Five of the fifteen machine learning algorithms evaluated were rule based learners. The paper reported very high average precision, recall, and accuracy for both software projects: 89% and 98% average precision, 83% and 99% average recall, and 87% and 96% average accuracy.

While other related work focused on evaluating different machine learning algorithms, a 2008 paper [19] introduced a benchmark for evaluating and comparing static analysis report prioritization and classification techniques. They evaluated the benchmark with three classification methods: one based on static analysis message type, one based on static analysis message locality within source code, and one based on a combination of the previous two. The benchmark, which consisted of static code analysis reports from six software projects, resulted in average recall values of 42%, 25%, and 32% for the three respective methods employed when used to perform classification. The study also provided precision and accuracy metrics in addition to the recall metric. The methods presented in this study include incremental analyst feedback which is beyond the scope of the work presented in this thesis.

Not much previous work marrying association rules to static code analysis error reports has been published. A 2006 paper [31] employed association rule learning to predict fault associations of the type “if fault x occurs then fault y is likely to occur”. The study compared association

rule learning with three other machine learning algorithms: PART, C4.5, and Naive Bayes and concluded that association rules perform roughly 25% better in terms of accuracy than the other methods. While the application of association rules in [31] did not focus on static code analysis report mining and is only loosely related to the work presented here, the study shows that association rules can be a powerful tool for gaining insight into the relationships between faults.

Much of the previous work focusing on machine learning methods as applied to static analysis reports employed features extracted from sources other than static analysis reports [28], [19], such as issue tracking systems (historical metrics such as how many faults have been fixed in this file/function) and versioning/revision control systems (change metrics such as code churn and number of developers having worked on a file/function). The work presented here is based solely on features extracted from static analysis reports and includes discussion and analysis on how the TP/FP distribution across source code can affect classification performance.

# Chapter 3

# Evaluation of Static Code Analysis Tools

This chapter presents the background, methodology, and results for the static code analysis tool evaluation. This chapter also includes threats to validity and the conclusion for the evaluation part of this thesis.

## 3.1 Methodology

The methodology for this chapter is based on automatic computation of confusion matrices. The following steps outline the way results were obtained:

- Run the benchmark through each of the three tools.
- Take the static analysis reports from each of the three tools and run them through an automated system of scripts to compute confusion matrices.

The static code analysis tools employed in this experiment were all commercial off the shelf products. One of the tools was designed with the explicit purpose of detecting security vulnerabilities in source code while the other two are static code analysis tools that have added the ability to detect vulnerabilities.

### 3.1.1 Common Weakness Enumeration and the Juliet Benchmark

In order to take repeatable and objective performance measurements from the selected tools a standard and easily reproducible way of categorizing software vulnerabilities was required. This

section provides some background information on the Common Weakness Enumeration [9] taxonomy regarding its design and hierarchical structure. The Juliet test suite [23] is also described along with a brief explanation of the structure of its test cases.

### CWE description

The Common Weakness Enumeration taxonomy is maintained by the MITRE Corporation with support from the DHS [9] and it provides a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities. Many static code analysis tools map their messages to CWEs. Each individual CWE ID represents a single vulnerability type or category. For example CWE 121 is Stack-based Buffer Overflow, CWE 78 is OS Command Injection, and so on. The CWE nomenclature consists of a hierarchical structure with broad category CWEs on the top level. The top level CWEs may have multiple children which in turn may or may not have further children. Each CWE may have one or more parents and zero or more children. The further down this hierarchy one goes the more specific the vulnerabilities become.

### Juliet Description

The Juliet test suites were created by the NSA evaluation effort [15] and have been made publicly available at the NIST web site [23]. The Juliet test suites were specifically designed to evaluate static code analysis tools by acting as input to such tools. Juliet consists of many sets (one set per CWE) of mini-tests, each with only one vulnerability mapped to one CWE. The test cases are designed in such a way as to produce output which can easily be processed automatically. This is accomplished by having a strict naming convention for functions and methods as well as by each test case having a standard design common throughout the entire test suite. Juliet test suite v.1.1 [23] was used for this work. For C/C++ it has 57,099 test cases for a total of 119 CWEs and for Java it has 23,957 test cases for 113 CWEs. Only those CWEs which were covered by all tools were included.

Juliets test case design is such that automatic evaluation of static analysis results can be feasibly implemented. Note that throughout the rest of this chapter the term function is used when referring to either C/C++ functions or Java methods. The main characteristic of Juliet is that correct code

constructs are found in functions which have the string ”good” in their names and incorrect code constructs are found in functions which have the string ”bad” in their name. This makes it so that a confusion matrix of true positives, false positives, true negatives, and false negatives can be automatically constructed as described in Section 3.1.2. This is a very simplified description of the general Juliet test case, which in addition to good and bad functions, also contains source, sink, and helper functions. The following is a detailed description of the Juliet test case format:

- All C test cases and the vast majority of C++ and Java test cases implement flaws contained in arbitrary functions. These test cases are called non-class-based flaw test cases. The simplest of these test cases consist of a single source file, while the more complex test cases are often implemented within multiple files.

- Each non-class-based flaw test case contains exactly one primary bad function. For test cases which span multiple files this primary bad function is located in the primary file.

- Each non-class-based flaw test case contains exactly one primary good function. This primary good function does not contain non-flawed constructs, its only purpose is to call the secondary good functions.

- Each non-class-based flaw test case contains one or more secondary good functions. Simpler test cases will have the non-flawed construct in the secondary good function(s), however in more complex test cases the secondary good function(s) can also be used to call some source and/or sink functions.

- Non-class-based flaw test cases which test for different data flow variants use source and sink functions. These source and sink functions can be either good or bad. Each such function can be mapped to either the primary bad function or exactly one secondary good function.

- When the implementation of a flaw cannot be represented within a single function, helper functions are used. Helper functions are mapped to either the bad function or one of the good functions. The source and sink functions mentioned above are not considered helper functions.

- Very few C++ and Java test cases implement vulnerabilities which are based on entire classes as opposed to individual statements or blocks of code. These are called class-based flaw test cases and are implemented across multiple files with each file containing a separate class:
- Each class-based flaw test case has one bad file. This file contains the required bad function.
- Each class-based flaw test case has one good file. This file contains a primary good function as well as one or more secondary good functions.
- There are some test cases for certain flow variants for C++ and Java which use virtual functions, in the case of C++, or abstract methods, in the case of Java. These test cases can be comprised of four or five files.
- Very few of the test cases contain only flawed constructs. As Juliet was being designed it was found that non-flawed constructs which correctly rectify the flaw being tested could not be generated for a small number of non-class-based vulnerabilities, which resulted in these bad-only test cases.

### 3.1.2 Description of the program for automatic computation of the tools performance

To compute the metrics for quantitative evaluation of the tools performance we implemented several scripts to automatically compute the confusion matrix for each CWE (i.e., the number of true positives (TP), false negatives (FN), true negatives (TN) and false positives (FP)). This automatic system consists of three steps shown in Figure 3.1.

The first step is to convert the tool report into a common format. This was done so that the same scripts could be used for all three tools without modification. The common format is a simple flat text file containing one report message per line. Each line is a comma-separated value list containing information such as file name, function name, line number, vulnerability CWE ID, and a unique identifier (in some cases more than one of the same vulnerability can be reported in the same place).

The second step is to parse each CWE directory in Juliet and assemble a complete list of test cases to be taken into account. This is needed because each CWE directory can have one or more

(thousands in some cases) test cases. Furthermore, for each test case considered, the script parses the test case source files to obtain the function list, which may be rather complex as described in section 3.1.1. This step is specific to the Juliet test suite and is common for the evaluation of all three tools.

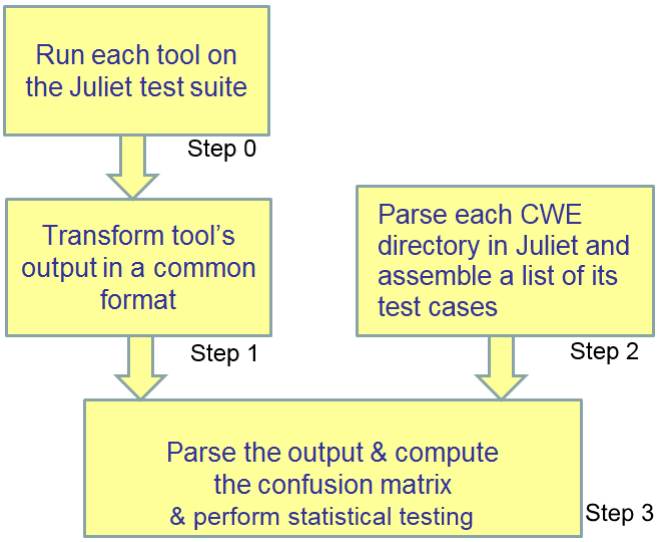



Figure 3.1: Metrics computation system block diagram

The third step loads the tool output file and iterates through it while matching the messages to the list of test cases assembled in the first step. The output of the second step is a flat text file (tr_out.txt) containing a list of each test case, the functions found within each test case (excluding the primary good function), and the messages produced by the tool for each test case.

During this third step all messages from some test cases were excluded, as well as some messages from all test cases. Following the suggestions given in [26] test cases and/or messages were discarded due to the following reasons:

- Messages produced by the tools which were not related to security vulnerabilities were excluded as the scope of the work described in this chapter was to evaluate tools performance regarding security vulnerability detection. In addition, almost all Juliet test cases contain incidental flaws, such as unreachable code and unused variables, which are typically considered unrelated to security vulnerabilities. Therefore such incidental flaws were also ignored, with the exception of two CWEs specifically designed for unreachable code, CWE561, and unused variables, CWE563.

- In the event that a test case contains more than one secondary good function as well as one or more helper function, the helper function cannot be mapped to the secondary good function from which it was called. This is an artifact of the automatically generated nature of the Juliet test suite. Due to this issue messages flagged within helper functions were ignored.
- Class-based flaw test cases comprise a very small subset of the total number of test cases. Furthermore their naming convention clashes with the naming convention of the much more numerous virtual function/abstract method test cases. Due to these issues class-based flaw test cases were ignored.
- Bad-only test cases can contain empty primary/secondary good functions but since they do not contain non-flawed constructs they cannot be counted as false positives and/or true negatives. All bad-only test cases were therefore excluded from the analysis as the results they produce cannot be properly evaluated.
- Additionally Windows-only test cases were excluded from all experiments as the available installation of Tool C was Linux-only. Only a few of the CWEs consist of Windows-only test cases.

The final step uses the text file output described above to compute the confusion matrix, shown in Table 3.1. After this final step statistical testing is performed on the results to draw empirical conclusions.

Table 3.1: Confusion matrix

| | Reported vulnerability | No message reported |
|---|---|---|
| Actual vulnerability | True Positives (TP) | False Negatives (FN) |
| No vulnerability (good function/method) | False Positives (FP) | True Negatives (TN) |

For each test case, the script parses through the messages associated with it and evaluates whether or not each message constitutes a true positive (TP), false negative (FN), true negative

(TN), false positive (FP), or if it should be ignored. The following points provide an explicit listing of which messages are taken into account and how special functions (i.e., source, sink) were taken care of when computing the confusion matrix:

- If a message was flagged as an appropriate CWE within bad code one true positive (TP) was counted.
- If no messages were flagged as an appropriate CWE within bad code one false negative (FN) was counted.
- If a message was flagged as an appropriate CWE within good code one false positive (FP) was counted.
- If no messages were flagged as an appropriate CWE within good code one true negative (TN) was counted.
- Source and sink functions, as described in section 3.1.1, were mapped to their calling function, either a primary bad function or a secondary good function. Messages flagged within a source or sink function were counted as messages relating to the specific calling function.
- Messages flagged in non-class-based test cases were all taken into account, with the exception of what is described in the Excluded Messages/Test-cases section.
- Messages flagged in virtual function/abstract method test cases were all taken into account, with the exception of what is described in the Excluded Messages/Test-cases section.
- For each bad function (one per test case) only one true positive was counted regardless of how many messages match the criteria for true positives.

The determination of whether or not a message was flagged as the appropriate CWE being tested goes beyond a simple exact match. As mentioned in section 3.1.1, each CWE may have one or more parents and zero or more children. Due to the nature of the CWE hierarchy and the complex relationships it establishes between individual CWEs the definition of “appropriate CWE” was expanded to include the currently investigated CWEs immediate parents and children. This was done because the vague relationships between CWEs, at the very least, make the problem

of automatic labeling of individual messages by the static analysis tools ambiguous enough to introduce some uncertainty in those labels. Without having access to the process by which static analysis tools label the reports they generate the benefits gained in terms of better chances for detecting true positives outweigh the slight bias introduced in favor of the tools.

A simple example to illustrate the above is as follows: consider Java CWE 129 and the associated list of its parents and children: CWE-867 (2011 Top 25 - Weaknesses On the Cusp), CWE-802 (2010 Top 25 - Risky Resource Management), CWE-20 (Improper Input Validation), CWE-189 (Numeric Errors), CWE-633 (Weaknesses that Affect Memory), CWE-738 (CERT C Secure Coding Section 04 - Integers ), CWE-740 (CERT C Secure Coding Section 06 - Arrays ), CWE-872 (CERT C++ Secure Coding Section 04 - Integers ), CWE-874 (CERT C++ Secure Coding Section 06 - Arrays and the STL), CWE-970 (SFP Secondary Cluster: Faulty Buffer Access). While computing the confusion matrix for CWE 129 a message flagged as any of the CWEs listed previously is considered an "appropriate CWE". Please note that only the immediate parent and children CWEs of the current CWE being looked at are considered. "Siblings" of the current CWE are not considered. For example, another one of the Java CWEs is CWE 190. It is also a child of CWE 189, just like CWE 129, however it is not considered when computing the confusion matrix for CWE 129.

The total number of bad functions and the total number of good functions counted by the first script were used as a sanity check. This was done because TP + FN = number of bad functions and FP + TN = number of good functions.

### 3.1.3 Metrics used for quantitative comparison of tools performance

For each CWE the script automatically produced the confusion matrix, which was then used to compute the following metrics used to quantitatively evaluate the static code analysis tools.

Accuracy: This metric provides the percentage of correctly classified instances, both true positives and true negatives, or the ratio of the sum of true positives and true negatives to the sum of true positives, true negatives, false positives, and false negatives.

$$\text{Acc} = \frac{(TN + TP)}{(TN + FN + FP + TP)} \tag{3.1}$$

Probability of Detection: Sometimes called recall, this metric provides the probability of correctly classifying true positives, or in other words the ratio of true positives to the sum of true positives and false negatives. The closer the probability of detection is to a value of 1 for a given CWE, the better the tool performs on that CWE.

$$\text{PD} = \frac{TP}{(FN + TP)} \tag{3.2}$$

Probability of False Alarm: This metric provides the probability of incorrectly classifying non-flawed constructs as flawed, or the ratio of false positives to the sum of true negatives and false positives. The closer the probability of false alarm is to 0 for a given CWE, the better the tool performs on that CWE.

$$\text{PF} = \frac{FP}{(TN + FP)} \tag{3.3}$$

Balance: Balance is a rather useful metric of performance because it combines the two most important metrics above: probability of detection and probability of false alarm. It is defined as the Euclidian distance from the ideal point of pf = 0, pd = 1 to a pair of (pf, pd). For convenience, the balance is normalized by the maximum possible distance across the ROC square (i.e., $\sqrt{2}$) and then subtracted from 1. It follows that higher balance is better since (pf, pd) point falls closer to the ideal point (0, 1).

$$\text{Bal} = 1 - \frac{\sqrt{(0 - PF)^2 + (1 - PD)^2}}{\sqrt{2}} \tag{3.4}$$

G-Score: G-Score is the harmonic mean between recall, (PD), and (1-PF). Similarly to balance, it helps elucidate the relationship between recall and probability of false alarm. G-Score values are close to 1 when recall is high and probability of false alarm is low, which is the desirable outcome.

$$\text{G-Score} = \frac{2PD(1 - PF)}{PD + 1 - PF} \tag{3.5}$$

### 3.1.4 Background on Statistical Testing

The goal of any statistical testing is to refute the given null hypothesis. In the case being presented here the null hypothesis was that there is no difference between the performance of the

three tools, where performance is defined in terms of the metrics described in Section 3.1.3. The experiment presented in this chapter is of a simple two-factor design (tools and CWEs), however because of the same samples being used for testing all of the tools, only one factor can be tested. The factor in question is the tool being tested, of which there are three. The Friedman rank sum test is a non-parametric version of the repeated measure ANOVA and is similar to the Kruskal-Wallis test in its use of ranks. The Friedman test process consists of ranking each data row (in this case the metric values per CWE) followed by a comparison of the rank values by column (in this case the columns represent each of the three tools tested). When the null hypothesis that there is no difference between tools' performance is refuted, post-hoc tests are used to determine where the difference is.

### 3.1.5 Quantitative evaluation based on real open source programs

Using the Juliet test suite for evaluation has several advantages. It allows automatic evaluation of the tools performance using metrics such as recall, probability of false alarm and balance, on a large number of test cases for many different vulnerabilities. However, individual Juliet test cases are rather small, containing only a single vulnerability of a specific type. Such synthetic test cases may not leverage the full potential of any given tool. Therefore the tools performance was also evaluated using real open source programs with known vulnerabilities in an attempt to discern any difference in performance.

In order to test static analysis on real programs software with known vulnerabilities for which exact locations in source code is also known was required. Furthermore fixed versions of the same software were needed in order to properly compare the tools. With these requirements in mind for evaluation of the selected tools we constructed a small test suite of three open source programs: Gzip, Dovecot, and Tomcat. The basic facts about these programs are given in Table 3.2.

This approach allowed evaluating the tools' performance on applications with realistic complexity. However it also has some limitations. Since unknown vulnerabilities may still be present in the source code, the assessment of all false negatives is impossible. Instead, only the messages relating to the locations of the known vulnerabilities were analyzed.

Each of the known vulnerabilities in Gzip and Dovecot was located within a single source file.

Table 3.2: Basic facts about open source programs used for evaluation of the static code analysis tools

| Name | Functionality | LOC | Lang | # of known vulnerabilities | Version with known vulnerabilities | Version with fixed vulnerabilities |
|---|---|---|---|---|---|---|
| Gzip | Archiving tool | ~8,500 | C | 4 | 1.3.5 | 1.3.6 |
| Dovecot | IMAP/POP3 server | ~280,000 | C | 8 | 1.2.0 | 1.2.17 |
| Tomcat | Java Servlet/JavaServer Pages implementation | ~4,800,000 | Java | 32 | 5.5.13 | 5.5.33 |

Each of the Tomcat vulnerabilities spanned several locations within the same file and/or multiple files. Specifically, out of 32 known vulnerabilities in Tomcat version 5.5.13, four occurred in one location within one file, nine occurred at multiple locations within one file, and 19 occurred across multiple files. A true positive was counted, i.e., vulnerability being detected, if at least one of the file(s)/location(s) were matched by a tool's message.

With a total of 44 known vulnerabilities between all three applications the breakdown in terms of CWE mapping is as follows:

- 10 vulnerabilities did not have available CWE mappings
- 8 vulnerabilities were mapped to CWE 264 (Permissions, privileges, and access controls)
- 7 vulnerabilities were mapped to CWE 22 (Path traversal)
- 7 vulnerabilities were mapped to CWE 200 (Information exposure)
- 5 vulnerabilities were mapped to CWE 79 (Cross-site scripting)
- 4 vulnerabilities were mapped to CWE 20 (Improper input validation)

- 1 vulnerability was mapped to CWE 399 (Resource management errors)
- 1 vulnerability was mapped to CWE 119 (Improper restriction of operations within the bounds of a memory buffer)
- 1 vulnerability was mapped to CWE 16 (Configuration)

Out of the CWEs listed above, two are part of MITREs Top 25 Most Dangerous Software Errors, namely CWE 22 (Path traversal) and CWE 79 (Cross-site scripting). These two CWEs add up to a total of 12 out of 44 known vulnerabilities being in MITREs Top 25. Since the number of known vulnerabilities was small, matching the messages to the vulnerabilities was done manually. It should be noted that due to the large number of messages produced by the tools, (see Table 3.7) analyzing all messages and inspecting the code to find the false positives was impractical.

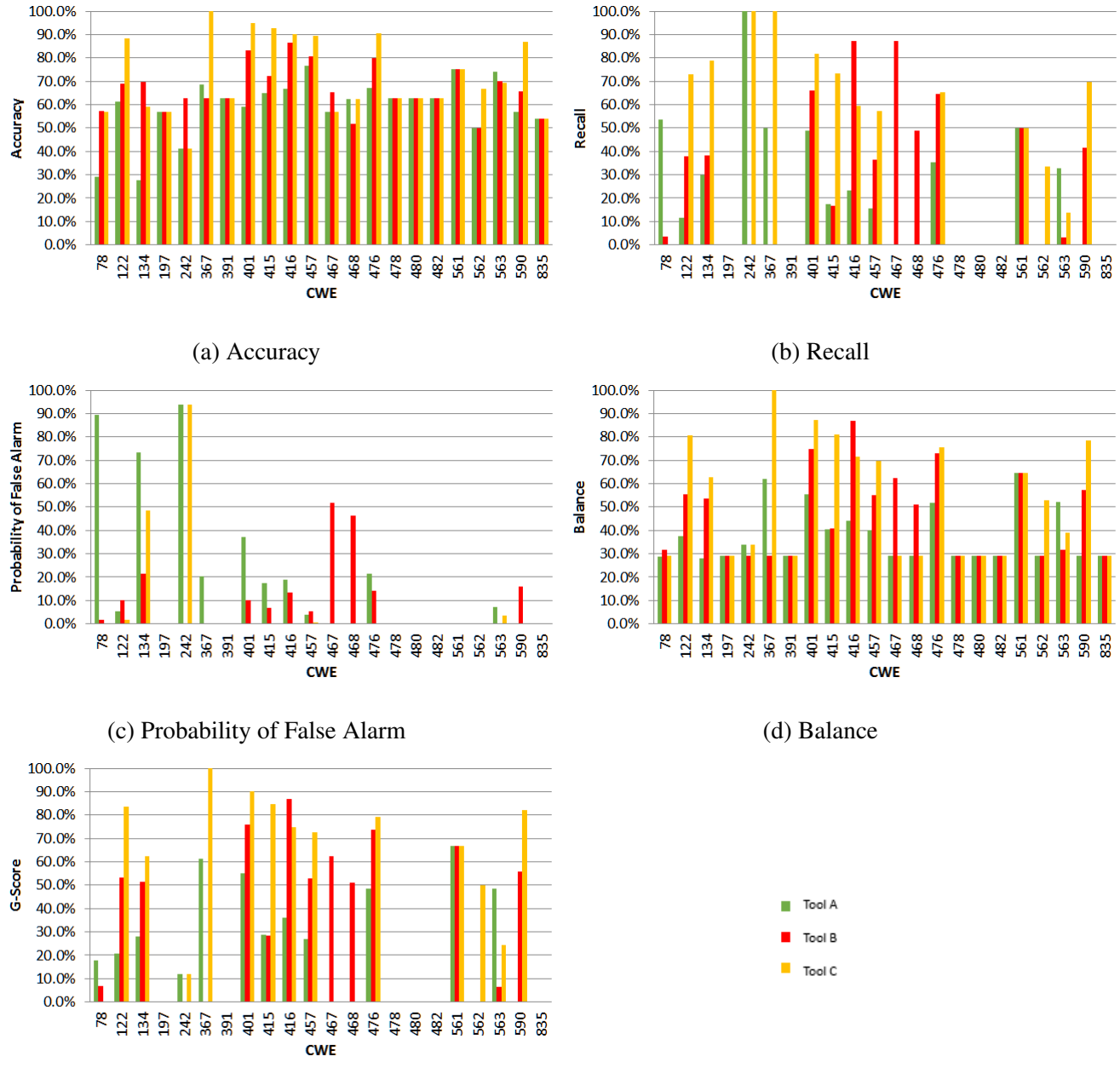
## 3.2 Evaluation Results based on Juliet Benchmark

In this section the quantitative results of the evaluation for all three tools with C/C++ and Java CWEs (i.e., 22 C/C++ CWEs common among all tools and 19 Java CWEs common among all tools) are presented. The tools' performance is then discussed with regards to several CWE categories and finally the results for the real-software-based evaluation are presented. The actual data used in this section is given in more detail in Appendix A.

### 3.2.1 Evaluation on C/C++ CWEs

Figure 3.2a presents the accuracy of the three tools for the C/C++ test suite. With the accuracy metric, the higher the value, the better the performance. From Figure 3.2a it can be observed that the three tools have similar performance with respect to accuracy, with Tool C performing slightly better.

Figure 3.2b is the tools recall values for the C/C++ CWEs. The recall metric (i.e., probability of detection) is important because it is related to the number of true positives (i.e., flawed constructs detected). As with accuracy, a higher recall value indicates better performance. It can be observed from Figure 3.2b that each tool has recall of 0% for several CWEs. Furthermore, for some CWEs,




(a) Accuracy

(b) Recall

(c) Probability of False Alarm

(d) Balance

(e) G-Score

Figure 3.2: Metrics for the C/C++ CWEs

such as 478 and 480, all three tools have recall of 0%. These recall values show that even though the accuracy may look promising, all of the tools perform rather poorly when it comes to detecting certain flawed constructs.

Figure 3.2c displays the values for the probability of false alarm in analyzing the C/C++ CWEs.

Here smaller values indicate better performance as they represent less non-flawed constructs being misclassified as flawed. As it can be observed from Figure 3.2c, Tool C appears to have comparatively lower false positive rate than Tool A and Tool B for all CWEs except CWE 134 and CWE 242.

Figure 3.2d presents the balance metric for the C/C++ CWEs. When it comes to the balance metric higher values indicate better performance. As it can be seen in Figure 3.2d the balance values for many CWEs were around 30%. This indicates rather poor overall performance from all tools. Similarly, as with accuracy, Tool C performed slightly better than the other two tool, most likely due to its low false positive rate. It also appears from Figure3.2d that Tool B performed slightly better than Tool A when it comes to this metric.

Figure 3.2e shows the G-Score metric for the C/C++ CWEs. Similar to balance, G-Score values close to 1 indicate performance with high recall and low probability of false alarm. As with the other metrics, all three tools display poor performance overall.

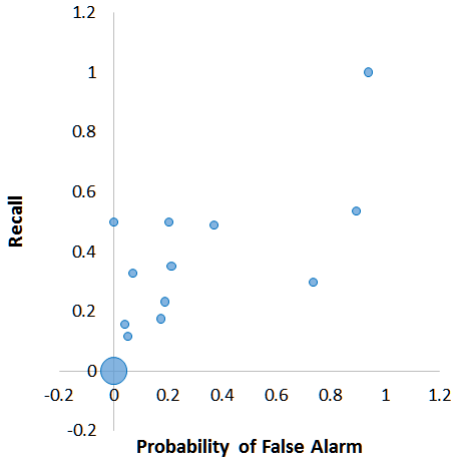
Figures 3.3a through 3.3c present the ROC squares for each of the three tools. In these figures the probability of false alarm is shown on the x-axis, while recall (i.e., probability of detection) is shown on the y-axis. The closer a point is to the ideal point (0, 1), the better the tools performance is on that particular CWE.

Figures 3.3a through 3.3c indicate the following main observations:

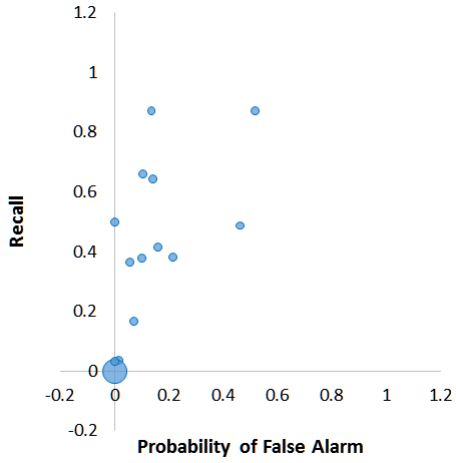
- For each of the three tools not many points are close to the ideal (0, 1) point.
- Tool C has a noticeably lower probability of false alarm than both Tool A and Tool B.
- For each tool there are multiple CWEs at the (0, 0) point as a consequence of the tools failing to detect any of the test cases associated with those CWEs.

### 3.2.2 Evaluation on Java CWEs

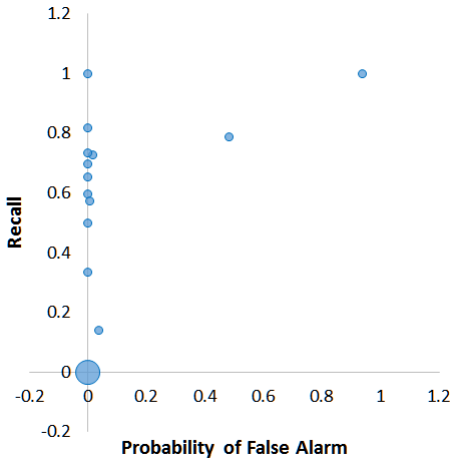
Figure 3.4a shows the accuracy metric for the Java CWEs. From Figure 3.4a one notices that the accuracy values for Java vary somewhat more than those for C/C++ shown in Figure 3.2a. Furthermore it is observed that all three tools attain a maximum accuracy value for several CWEs. This means that for those CWEs and the respective tool(s), all constructs, both flawed and non-




(a) ROC square for the C/C++ CWEs with Tool A




(b) ROC square for the C/C++ CWEs with Tool B



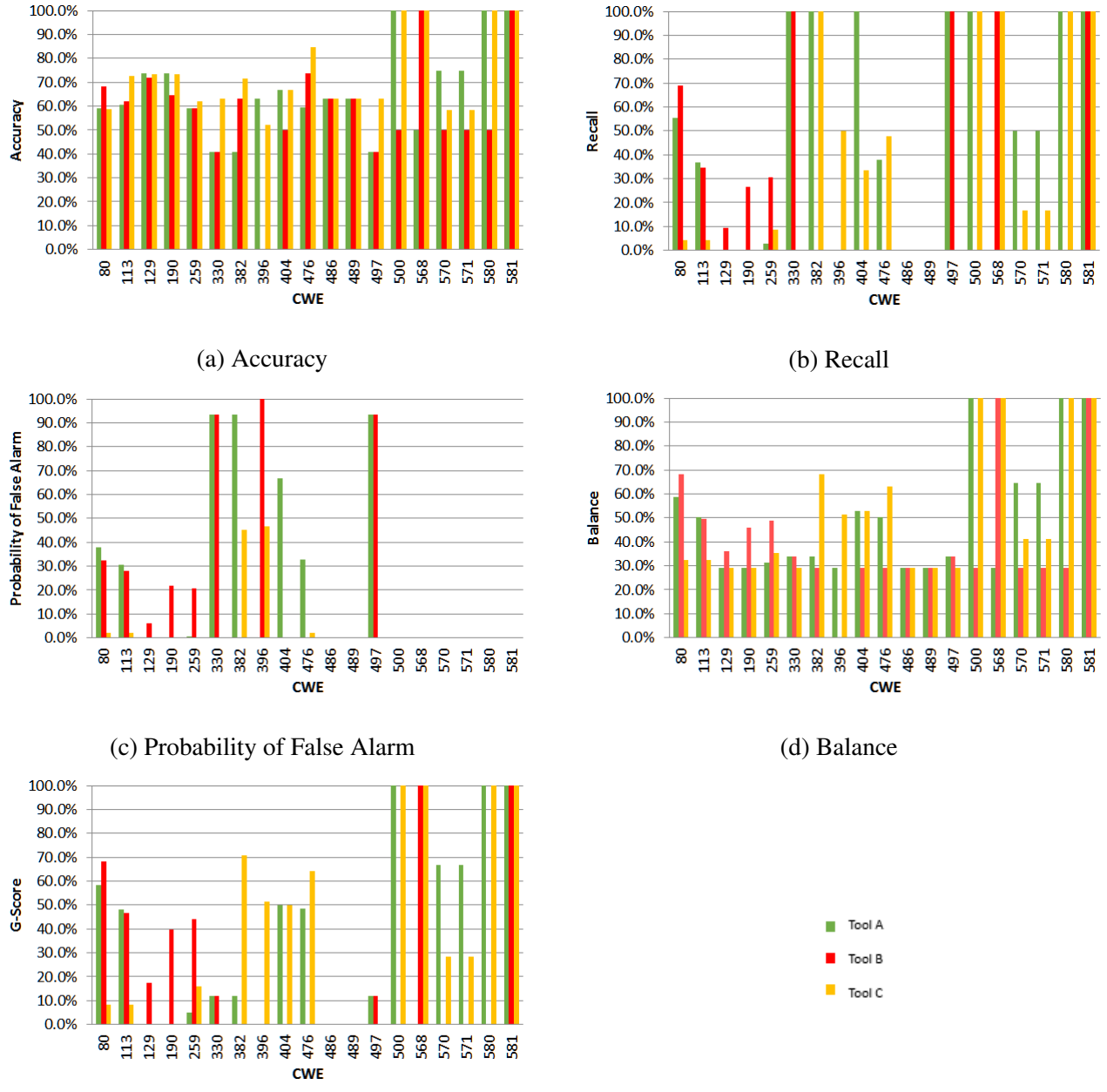

(c) ROC square for the C/C++ CWEs with Tool C

Figure 3.3: ROC squares for the C/C++ CWEs. Some of the data points above represent multiple points stacked on top of one another, for example many CWEs cluster at point (0,0).

flawed, were correctly classified. As in case of the C/C++ CWEs, Tool C seems to be performing slightly better than the other two tools

Figure 3.4b presents the recall values for the Java CWEs. One can see that there are again a couple CWEs (486 and 489) for which none of the tools correctly flagged any flawed constructs, however there were not as many as in the case of the C/C++ suite.

Figure 3.4c displays the probability of false alarm values for the Java CWEs. The trend here is similar to the C/C++ false alarm values; Tool C performed significantly better than the other two tools and Tool A performed slightly better than Tool B.




(a) Accuracy

(b) Recall

(c) Probability of False Alarm

(d) Balance

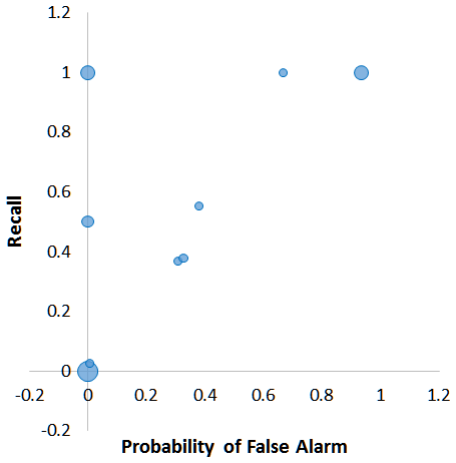
(e) G-Score

Figure 3.4: Metrics for the Java CWEs

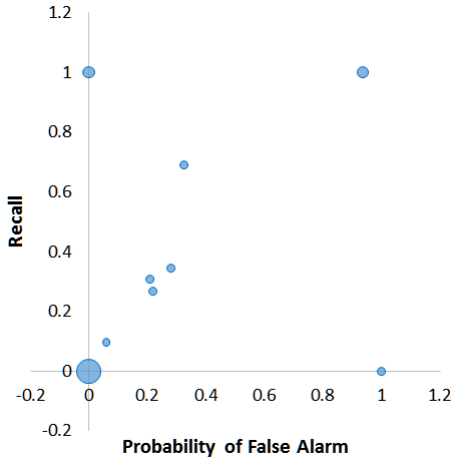
Figure 3.4d shows the balance values for the Java CWEs. Again one notices a similar trend to the C/C++ balance values for many CWEs the balance values were around 30% for either all or some of the tools, which is an indication of overall poor performance. Tool C and Tool A appear to perform slightly better than Tool B.

Figure 3.4e shows the G-Score values for the Java CWEs. As with the C/C++ G-Score values, the Java data shows overall poor performance from all three of the tools.

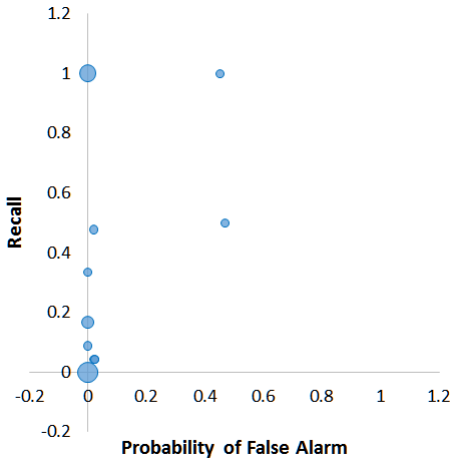
Figures 3.5a through 3.5c show the ROC squares for the Java CWEs with Tool A, Tool B, and Tool C, respectively. The closer the points are to the ideal point (0, 1), the better the tools performance.




(a) ROC square for the Java CWEs with Tool A




(b) ROC square for the Java CWEs with Tool B




(c) ROC square for the Java CWEs with Tool C

Figure 3.5: ROC squares for the Java CWEs. Some of the data points above represent multiple points stacked on top of one another, for example many CWEs cluster at point (0,0).

Based on Figures 3.5a through 3.5c one can make following main observations:

- For each of the three tools not many points are close to the ideal (0, 1) point.

- Tool C has noticeable smaller probability of false alarm than Tool A and Tool B.
- For each tool there are multiple CWEs at the (0, 0) point as a consequence of failing to detect any of the test cases associated with those CWEs.

### 3.2.3 Summary of the tools performance on the C/C++ and Java subsets

Table 3.3 presents the summary results of how each tool performed with respect to the 22 C/C++ CWEs and 19 Java CWEs. For each metric four statistic measures are provided: mean (simple arithmetic average), median, a weighted mean which takes into account the number of test cases for each CWE (i.e., CWEs with larger numbers of test cases carry more weight than ones with only a few test cases), and an overall value computed from the total sum of TP, FP, TN, and FN over all CWEs (that is, the sums of all TPs, FPs, TNs, and FNs for all CWEs). The weight for each CWE was simply computed as follows:

$$\text{Weight} = \frac{\text{number of test cases for current CWE}}{\text{total number of test cases}} \tag{3.6}$$

The best values for each tool are shown in grey. As it can be seen from Table 3.3, Tool C tends to perform slightly better than the other tools for most metrics, except for the recall for the Java CWEs for which Tool A performs best. However, Tool C still has slightly better Balance statistics than Tool A, which is due to lower probability of false alarm.

As it can be seen from Figures 3.2a to 3.2e and Figures 3.4a to 3.4e, none of the tools was able to detect all vulnerabilities. Out of the 22 C/C++ CWEs, 9 were detected by all three tools (around 41%) and 6 were not detected by any tool (i.e., 27%). The remaining 7 CWEs were detected by a single tool or a combination of two tools.

Table 3.3: Accuracy, Recall, Probability of False Alarm, Balance, and G-Score for both C/C++ and Java CWEs with all three tools

| | | | Tool A | Tool B | Tool C |
|---|---|---|---|---|---|
| Accuracy | C/C++ | Mean | 59% | 67% | 72% |
| | | Median | 63% | 64% | 64% |
| | | Weighted | 51% | 69% | 82% |
| | | Overall | 51% | 69% | 82% |
| Accuracy | Java | Mean | 67% | 60% | 73% |
| | | Median | 63% | 63% | 67% |
| | | Weighted | 69% | 66% | 72% |
| | | Overall | 69% | 66% | 73% |
| Recall | C/C++ | Mean | 21% | 26% | 39% |
| | | Median | 14% | 10% | 42% |
| | | Weighted | 26% | 34% | 61% |
| | | Overall | 25% | 32% | 59% |
| | Java | Mean | 49% | 35% | 36% |
| | | Median | 50% | 18% | 17% |
| | | Weighted | 13% | 25% | 3% |
| | | Overall | 14% | 26% | 3% |
| Prob. of False Alarm | C/C++ | Mean | 18% | 9% | 7% |
| | | Median | 2% | 1% | 0% |
| | | Weighted | 35% | 11% | 6% |
| | | Overall | 35% | 12% | 7% |
| | Java | Mean | 24% | 25% | 5% |
| | | Median | 0% | 3% | 0% |
| | | Weighted | 10% | 19% | 1% |
| | | Overall | 10% | 19% | 1% |

Continued on the next page

Table 3.3: (continued)

| | | | Tool A | Tool B | Tool C |
|---|---|---|---|---|---|
| Balance | C/C++ | Mean | 39% | 46% | 53% |
| | | Median | 29% | 36% | 46% |
| | | Weighted | 36% | 52% | 70% |
| | | Overall | 42% | 51% | 71% |
| | Java | Mean | 50% | 43% | 52% |
| | | Median | 34% | 34% | 41% |
| | | Weighted | 36% | 44% | 31% |
| | | Overall | 39% | 46% | 31% |
| G-Score | C/C++ | Mean | 20% | 31% | 40% |
| | | Median | 15% | 18% | 37% |
| | | Weighted | 24% | 45% | 68% |
| | | Overall | 36% | 47% | 73% |
| | Java | Mean | 36% | 23% | 38% |
| | | Median | 12% | 0% | 29% |
| | | Weighted | 16% | 35% | 4% |
| | | Overall | 24% | 39% | 6% |

The results obtained when running the Java CWEs were similar. None of the tools was able to detect all vulnerabilities and many CWEs were not detected by either tool. Out of the 19 CWEs, 4 were detected by all three tools and 2 by none (i.e., over 10%). The remaining 13 CWEs were detected by a single tool or a combination of two tools.

An important observation is that there is no significant difference in the recall values. Furthermore, the values themselves fall under 50% which would be the expected outcome at random guessing. This implies that the three tools are worse at detecting true positives than random chance. There does appear to be a difference in performance when looking at the probability of false alarm metric between tools A and C. This metric is important when trying to minimize false positives so it represents a positive result. A difference in performance between tools A and C is also found when looking at the accuracy metric as well but the accuracy metric is not considered to be very representative in the context of this experiment.

The results obtained were tested for significance using the Friedmann test. Furthermore, post-hoc testing was performed in those cases in which a significant difference was detected by the Friedmann test. The results are given in Table 3.4 and show the null hypotheses of no difference among tools to be rejected between tools A and C with respect to Accuracy, Balance, and Probability of False Alarm for C/C++ CWEs. The null hypotheses of no difference in performance among the three tools were not rejected for Java CWEs.

Table 3.4: Friedmann results for both C/C++ and Java CWEs with all three tools

| | Accuracy | Recall | Balance | Probability of False Alarm | G-Score |
|---|---|---|---|---|---|
| C/C++ Tools A, B, C | Reject null hypothesis | Fail to reject | Reject null hypothesis | Reject null hypothesis | Fail to reject |
| Difference | A and C | N/A | A and C | A and C | N/A |
| Java Tools A, B, C | Fail to reject | Fail to reject | Fail to reject | Fail to reject | Fail to reject |
| Difference | N/A | N/A | N/A | N/A | N/A |

### 3.2.4 Evaluation of tools performance by CWE category

Much of the related work groups software vulnerabilities in broad categories often defined by researchers themselves without a firm objective reason. While this approach may introduce new threats to validity, the idea of testing for security vulnerability detection based on categories should be explored. One tool may perform better than others on a specific type of vulnerability. In order to get a better idea of whether or not performance is affected by categorizing vulnerabilities, the CWEs for both C/C++ and Java were grouped according to the general vulnerability classes and categories found in [10]. These vulnerability categories are themselves CWEs, however unlike the CWEs so far discussed, these category/class CWEs serve no purpose other than to help bring structure to the CWE hierarchy. Using these top-level CWE categories is more objective, consistent, and repeatable than constructing categories and assigning what appear to be related CWEs to them. The resulting grouping was applied only to those CWEs which belonged to a group with

more than one CWE. For example, assuming CWE 121 is one of the CWEs investigated, if CWE 121 fits into the category CWE 970 and is the only CWE that fits in category CWE 970, then that category CWE 970 is excluded from the evaluation. The groupings explored for C/C++ are shown in Table 3.5.

Table 3.5: C/C++ CWE category grouping

| Category | CWE IDs |
|---|---|
| CWE 398 Indicator of poor code quality | 478, 561, 562, and 563 |
| CWE 399 Resource management errors | 401, 415, 416, and 590 |
| CWE 465 Pointer issues | 467, 468, and 476 |
| CWE 569 Expression issues | 480 and 482 |
| CWE 633 Weaknesses that affect memory | 122 and 134 |

Because the CWEs differed between the C/C++ and Java test suites the groupings that were used for Java are show in Table 3.6.

Table 3.6: Java CWE category grouping

| Category | CWE IDs |
|---|---|
| CWE 189 Numeric errors | 129 and 190 |
| CWE 254 Security features | 259 and 330 |
| CWE 399 Resource management errors | 404 and 568 |
| CWE 422 Web problems | 80 and 113 |
| CWE 569 Expression issues | 570 and 571 |

The recall, false alarm, and balance metrics were investigated with the groups outlined above using all three tools. Almost nothing of relevance was found; a few observed patterns only reinforced the previous results. One conclusion able to be drawn from looking at the probability of false alarm for both C/C++ and Java was that Tool C produces almost no false positives. This is consistent with the previous findings. Furthermore, from the C/C++ balance and recall metrics it also appears that Tool C performs better than the other two on CWE 399 Resource management errors (CWEs 401, 415, 416, and 590) and CWE 633 Weaknesses that affect memory (CWEs 122 and 134), both categories having to do with memory management and buffer issues. Looking at

the Java balance and recall it seems that Tool A performs relatively better than the other two on CWE 569 Expression issues (CWEs 570 and 571). Going back to C/C++ recall one can notice that all three tools failed to detect any true positives on CWE 569 Expression issues (CWEs 480 and 482).

## 3.3 Evaluation Results based on Real Software

The results of tools evaluation based on the open source programs shown in Table 3.7 confirm the evaluation done based on the Juliet test suite — static code analysis tools were not very successful in detecting the known vulnerabilities. Thus, in case of the smallest program, Gzip, Tool B did not identify any of the four vulnerabilities. Even though Tool A and Tool C detected one of the four known vulnerabilities, they both reported the same vulnerability in the fixed version, which are actually false positives. The results using Dovecot were even worse, none of the three tools detected any of the eight known vulnerabilities. Tool A was the most successful on Tomcat, the largest of the three programs used for evaluation; it detected seven out of the 32 known vulnerabilities compared to five vulnerabilities detected by Tool C and three by Tool B. The fact that Tool A was able to detect more known vulnerabilities than the other two tools for the Java application is also consistent with evaluation based on the Juliet test suite. Unfortunately, Tool A and Tool C also reported two and one vulnerabilities at the same locations in the fixed versions, respectively.

Table 3.7: Number of detected known vulnerabilities in the vulnerable versions (true positives) and reported vulnerabilities in the fixed versions (false positives)

| Program | Version | Tool A | | Tool B | | Tool C | |
|---|---|---|---|---|---|---|---|
| | | Messages | Det. Vulns | Messages | Det. Vulns | Messages | Det. Vulns |
| Gzip | 1.3.5 | 112 | 1/4 | 36 | 0/4 | 119 | 1/4 |
| | 1.3.6 | 206 | 1/4 | 125 | 0/4 | 374 | 1/4 |
| Dovecot | 1.2.0 | 8263 | 0/8 | 538 | 0/8 | 1356 | 0/8 |
| | 1.2.17 | 8655 | 0/8 | 539 | 0/8 | 1293 | 0/8 |
| Tomcat | 5.5.13 | 12399 | 7/32 | 12904 | 3/32 | 20608 | 5/32 |
| | 5.5.33 | 167837 | 2/32 | 13129 | 0/32 | 21128 | 1/32 |

## 3.4 Threats to Validity for Tools' Evaluation

Several threats to validity may have affected the empirical evaluations presented in this chapter. Similar concerns as the ones discussed here were discussed in the latest evaluation done by the NSA [15]. Next these threats are briefly discussed in terms of construct, internal, conclusion, and external threats along with the approaches taken to mitigate their effects on the results.

### 3.4.1 Construct Validity

Construct validity refers to ensuring that what is intended to be tested is actually tested for. One threat to construct validity is associated with the chosen static analysis tools employed in this work. The selected tools may not be a representative sample of the vast number of available static analysis tools. However, the results presented in this work are believed to be representative of the current state of the art and practice in static analysis as the three tools employed are widely used commercial tools, each with large user bases.

Another construct validity threat relates to how the CWEs reported by the three tools are matched to the CWEs found in the benchmark suite. Despite being hierarchical in nature, the CWE nomenclature is more similar to a directed graph than a tree-like structure. This is because CWEs may have many different parents and children. For this reason static code analysis tools may have a difficult time mapping their messages to CWEs. As an example consider a situation in which a static analysis tool has to decide whether to report a message as CWE A or CWE B, where CWE A and CWE B are parent and child, respectively. Since CWE A is the parent to CWE B, mapping the message to it would be a safe call, however as the parent CWE A is also less descriptive than CWE B for the same reason that makes it a safe call in the first place: CWE A is more general than CWE B. In the end, responsibility for mapping between messages and CWE IDs falls to individual tools and as such there is a risk that different tools will report different CWE IDs for the same piece of code. Due to the structure of the CWE nomenclature and the inherent inter-relatedness of security vulnerability types, an exact matching would be likely to underestimate the tools' detection performance. Previous studies employed groupings of CWEs but found that the groupings would have to be improved upon and that there is no straightforward way to group vulnerability types [26]. Thus when determining whether or not the CWE reported by a tool

matched the specific CWE being tested for the hierarchical relationships established by the CWE nomenclature were used as outlined in the Methodology section. This was done in order to avoid both a strict exact matching, which might underestimate the tools' performance, and loose groups containing tenuously related vulnerabilities, which might overestimate the tools' performance.

Static analysis tools' performance is highly dependent on the input source code and the amount and type of defects found in that code. Employing a publicly available benchmark suite covering a wide array of different defects makes the experimental approach repeatable. However, given the benchmark's synthetically generated nature, three open source projects with known security defects were included to gage the tools' performance with real vulnerabilities.

### 3.4.2 Internal Validity

Internal validity is concerned with any influences that may affect measurements and independent variables. With static analysis tools reporting very large numbers of messages, analysts and developers must manually examine and classify messages as either true or false positives. Clearly human intervention introduces the risk of misclassification which constitutes a threat to internal validity. To mitigate this risk the Juliet benchmark was used which identifies the vulnerable code constructs, that is the bad functions, as well as the corresponding non-vulnerable code, the good functions. This in turn provides for the usage of automatic classification of static analysis messages and removes the internal validity threat of human error with respect to classification. The three open source case studies had a small number of known vulnerabilities and this allowed for manual evaluation of the tools' reports. Knowing the true outcome, that is the presence of known vulnerabilities in specific code locations for the versions with known defects and the lack of those vulnerabilities in the fixed versions, meant that no misclassifications occurred when inspecting the tools' output for the real software projects.

### 3.4.3 Conclusion Validity

Conclusion validity has to do with threats that influence the drawing of correct conclusions. This threat was mitigated by employing a proper statistical experimental design based on a benchmark test suite and correctly applying the non-parametric statistical test applicable to the data.

Another threat to conclusion validity is a small sample size. This threat was removed by using the benchmark test suite with numerous CWEs in both the C/C++ and Java cases as well as only applying the statistical tests on those CWEs with large numbers of test cases.

### 3.4.4 External Validity

External validity has to do with the generalizability of experimental results. An obvious threat to external validity is the synthetic nature of the benchmark's test cases. This threat was eliminated by the inclusion of three open source application case studies whose results confirmed the tools' performance with the benchmark suite.

## 3.5 Conclusion on Tools' Evaluation

The work presented in this chapter is focused on evaluation of static code analysis tools' performance with respect to the detection of security vulnerabilities. The experimental design based on a synthetically generated benchmark suite allowed for automatic evaluation of tools' performance, the quantitative assessment of tools' performance per CWE and across all CWEs, and for statistical evaluation of the results.

In spite of claims made by static analysis tool vendors and recent improvements in the static analysis field, the performance of static analysis tools with respect to security vulnerabilities is not satisfactory. A few CWEs were detected by all three tools while others were detected by a combination of two tools or by a single tool. However, many CWEs were completely missed by all three tools. Thus, none of the three tools reported 27% of the C/C++ CWEs and 11% of the Java CWEs. The tools' detection performance varied significantly by vulnerability type, however none of the tools significantly outperformed the other tools across all vulnerability types.

The results of the evaluation with real open source programs were consistent with the evaluation based on the Juliet test suite. By performing this part of the evaluation the conclusion that tools have high false negative rates (i.e., are not able to identify majority of the known vulnerabilities) was strengthened.

The results presented in this chapter were summarized in a research paper which was submitted for publication [17].

# Chapter 4

# Static Code Analysis Message Classification

This chapter presents the background, methodology, and results for the static code analysis message classification experiments. Threats to validity and the conclusion are also provided in this chapter.

## 4.1 Methodology

This section presents the methodology employed in the message classification chapter of this thesis. Briefly summarized the methodology is based on labeled classification of messages using incremental percentages of training data in order to emulate an on-going analysis effort over time.

Most machine learning algorithms operate on numerical data and therefore cannot be used in this context. The approach taken in this work is to use Classification Based on Associations (CBA) [25] on features directly extracted from static analysis reports. This algorithm was chosen because it operates on categorical data such as the fields in a static analysis message report. The CBA algorithm integrates classification with association rule mining in a supervised algorithm. Note that the class labels (TP, FP) are the analysts decisions for each message. While simple association rule mining strives to find as many rules as it can, classification rule mining instead attempts to construct a small set of rules that can be used as a classifier [25]. The algorithm is divided into two main steps: a rule generating step which produces a large set of association rules and a classifier building step which constructs the smaller rule based classifier. The initial research by Liu et al with CBA [25] showed that the resulting classifiers performed better than other algorithms that

operate on categorical datasets, such as C4.5. For the experiments presented in this thesis the CBA implementation available from the Weka machine learning framework [33] was used.

The features used for classification, which were extracted from each dataset's messages, are as follows:

- Filename: source code filename where the potentially faulty code is located
- Function: function or method where potentially faulty code is located
- Line #: line number(s) where faulty code is located
- Severity: severity ranking of the fault
- Checker: checker that flagged the fault
- TP/FP label: analyst's determination whether the message is TP or FP

An example of one such message instance is as follows:

FileName.java,main_method,168,Critical,SOME.CHECKER,FP

A typical data mining experiment as found in literature involves 10-fold cross validation. This means the given dataset is split into 10 equal folds (i.e., subsets), and then the training is performed on 9 of them, and the testing is performed on the remaining 10th fold. This is repeated 10 times, with each fold used exactly once for testing. The problem with this approach is that it simulates the situation in which an analyst has to have 90% of the analysis done before attempting to do predictive classification. This clearly has a limited practical value. Therefore, we conducted several experiments exploring the feasibility of employing the CBA algorithm on smaller subsets of labeled data. For each of the datasets, we explored the performance of CBA in classifying the messages as TP/FP when using 25%, 50%, 75%, and 90% subsets for training, and the remaining part for testing. For the 25% experiment, since a cross validation approach would not work, the experiment was performed using random stratified selection with four repetitions. The results from the four experiments were averaged together. The experiments for 50%, 75%, and 90% were performed using cross validation (that is, 2-fold, 4-fold, and 10-fold, respectively) and again the results were averaged.

Table 4.1 gives an overview of how confusion matrices were built. The top row denotes the classification given to messages by the CBA algorithm, while the left column denotes the actual classification of messages by analysts, which was used as a ground truth.

Table 4.1: Confusion matrix

| | Classified as TP | Classified as FP |
|---|---|---|
| Actual TP | True Positives (TP) | False Negatives (FN) |
| Actual FP | False Positives (FP) | True Negatives (TN) |

The recall and precision metrics are used to determine how successful the classification is. Recall, given with equation (4.1), or probability of detection, is defined as the number of correctly classified true positives (TP) divided by the total number of true positives (i.e., those classified as true positives (TP) and those misclassified as false negatives (FN) ). Precision, given with equation (4.2), is defined as the ratio between the number of correctly classified true positives (TP) and the total number of instances classified as true positives (i.e., the sum of true positives (TP) and false positives (FP) ).

$$\text{Recall} = \frac{\text{TP}}{\text{FN} + \text{TP}} \tag{4.1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.2}$$

In other words recall represents the percentage of true positives which are correctly classified as true positives while precision represents the percentage of instances classified as true positives which actually are true positives. Since Weka reports recall and precision rates for both classes, TP and FP, both of them are also reported here. However, the recall for the TP class is more important as it signifies how well the algorithm performs when predicting for true positives (i.e., real faults in source code), that is, not missing real existing faults.

## 4.2 Datasets

For the purposes of this empirical study the data required were analysis reports containing TP and FP determinations for individual messages produced by the static code analysis tools. In this

study datasets extracted from five NASA projects' software analyses were used. All datasets are the result of static analysis efforts performed with the same static code analysis tool. Two of the datasets, NASA_1 and NASA_2, consist of analysis data from large scale ground station software written mostly in C++ with some Java code. The other three datasets come from flight software for science missions and were all written in C++. It should be noted that the datasets used in this work consist of only those messages which have been manually analyzed. The number of total messages for the datasets presented may be much higher than the number of analyzed messages. The analysts determinations (either TP or FP) are taken as the "class label" of each message. This is required for two reasons. Firstly, studying the distribution of true and false positives would be impossible without such labels. Secondly, since CBA is a supervised algorithm it must be provided with class labels in order to determine a classification model. Note that the "Function" feature did not exist in the NASA_1 dataset and therefore only the remaining five features, as described in Section 4.1 were used for that dataset.

The basic information on the five datasets, annotated as NASA_1 to NASA_5, is given in Table 4.2 and includes the number of instances, the number and percentage of TP and FP messages, and the precision of the static analysis tool's classification. Of the datasets experimented with the NASA_2 and NASA_5 datasets are the more balanced, with 29-30% true positives and the remaining 70-71% of the messages being false positives. The NASA_1, NASA_3, and NASA_4 datasets are much more unbalanced, with only 4-5% true positives and 95-96% false positives.

Table 4.2: Basic information on the five datasets

| Project | # of Instances | # and % of TP | # and % of FP | Tool Classification Precision |
|---|---|---|---|---|
| NASA_1 | 726 | 34 (5%) | 692 (95%) | 4.7% |
| NASA_2 | 344 | 100 (29%) | 244 (71%) | 29.0% |
| NASA_3 | 375 | 20 (5%) | 355 (95%) | 5.3% |
| NASA_4 | 455 | 18 (4%) | 437 (96%) | 3.9% |
| NASA_5 | 144 | 42 (30%) | 102 (70%) | 29.2% |

## 4.3 Results of the Classification Experiments

The results of using CBA on all five datasets are given in Table 4.3. NASA_1 results are very good, with less than 0.5 value for TP recall when using only 25% of the data for training and all other with much higher recall for TP (0.88-0.91). This means that for NASA_1 analysts could run the CBA algorithm after labeling 50% of the messages as TP or FP and obtaining good predictions for remaining 50%, with TP recall of 0.9 and FP recall of 1. Interestingly, the predictions with 50% of labeled messages were as good as with 75% labeled messages.

One would intuitively expect the NASA_1 dataset to fail to produce good predictions given how unbalanced the dataset is, with only 5% TP messages (see Table 4.2). Closer look at the NASA_1 dataset showed that, even though NASA_1 had only 5% TP messages of all messages inspected by the analysts, the TP were very localized, i.e., found in only two files. Due to the localization of TP and the fact that they were often flagged by the same checkers, the data mining results are quite good.

To illustrate, the distributions of TP and FP messages per file as well as per checker are presented as bubble charts in Figure 4.1. The size of the bubbles represent either number of files or number of checkers with a given TP/FP ratio represented by the bubble's position on the chart. The bubble sizes have been scaled relative to one another for all per-file bubble plots and for all per-checker bubble plots. The reason for this is to allow quick and easy comparisons of per-file or per-checker bubble sizes across the five projects. Note that comparing the bubble sizes for a per-file plot with the bubble sizes on a per-checker plot is meaningless.

The localization phenomenon can be seen in Figure 4.1a; the "L" shape of the bubble plot indicates that files either contain true positives or false positives, but not a mix of the two, with the exception of one file. Figure 4.1b presents the bubble plot for TP/FP per checker. The most interesting observations in this figure are the three right most bubbles, which represent three checkers, each with a large number of only FP messages reported which leads to FP recall values of or close to 1. Figures 4.1a and 4.1b represent advantageous cases that allow the algorithm to generate very descriptive rules that lead to excellent predictions, even with 50% of the messages labeled. The large bubble at position (1,0) represents 192 singletons, that is, only one FP message in a file with no other messages flagged. So in the case of the NASA_1 dataset 192 files each resulted in only

one static code analysis message.

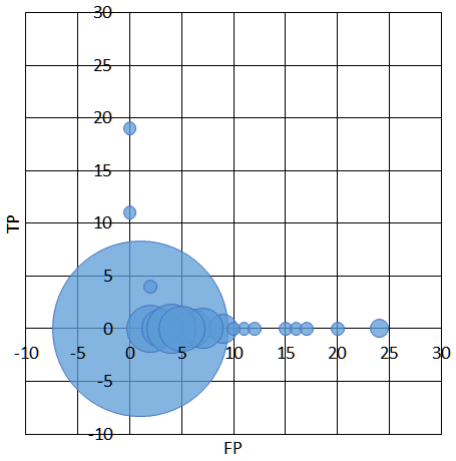
It should be noted that NASA_1 was the only dataset which resulted in rules with more than one feature on the left side. Such complex rules should be better predictors as they narrow down the classification criteria. The following example shows such a rule:

line=45 checker=SOME.CHECKER severity=Critical ==> result=TP

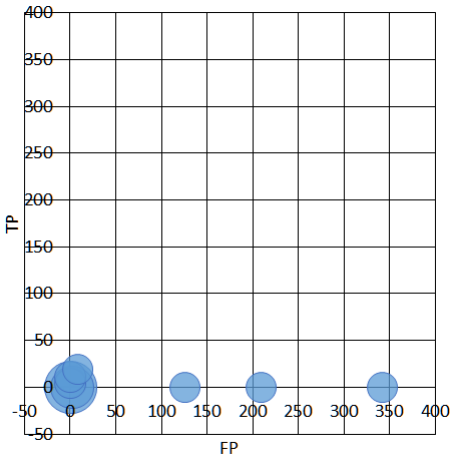
Table 4.3: CBA Learning Results for all datasets.

| Project | Experiment | TP Recall | FP Recall | TP Precision | FP Precision |
|---|---|---|---|---|---|
| NASA_1 | Train on 25%, test on 75% | 0.48 | 1 | 1 | 0.98 |
| | Train on 50%, test on 50% | 0.91 | 1 | 1 | 0.99 |
| | Train on 75%, test on 25% | 0.91 | 1 | 1 | 0.99 |
| | Train on 90%, test on 10% | 0.88 | 1 | 1 | 0.99 |
| NASA_2 | Train on 25%, test on 75% | 0.46 | 0.82 | 0.53 | 0.78 |
| | Train on 50%, test on 50% | 0.44 | 0.93 | 0.72 | 0.8 |
| | Train on 75%, test on 25% | 0.33 | 0.99 | 0.94 | 0.78 |
| | Train on 90%, test on 10% | 0.33 | 0.98 | 0.85 | 0.78 |
| NASA_3 | Train on 25%, test on 75% | 0 | 1 | 0 | 0.95 |
| | Train on 50%, test on 50% | 0 | 1 | 0 | 0.95 |
| | Train on 75%, test on 25% | 0.25 | 1 | 1 | 0.96 |
| | Train on 90%, test on 10% | 0.35 | 1 | 1 | 0.96 |
| NASA_4 | Train on 25%, test on 75% | 0 | 1 | 0 | 0.96 |
| | Train on 50%, test on 50% | 0.167 | 1 | 1 | 0.96 |
| | Train on 75%, test on 25% | 0.33 | 0.99 | 0.86 | 0.97 |
| | Train on 90%, test on 10% | 0.33 | 0.99 | 0.86 | 0.97 |
| NASA_5 | Train on 25%, test on 75% | 0.03 | 0.97 | 0.33 | 0.69 |
| | Train on 50%, test on 50% | 0.69 | 0.79 | 0.58 | 0.86 |
| | Train on 75%, test on 25% | 0.69 | 0.75 | 0.54 | 0.86 |
| | Train on 90%, test on 10% | 0.69 | 0.75 | 0.54 | 0.86 |

Unlike for NASA_1, the recall values for TP for NASA_2 dataset were less than 0.5 for all levels of labeled data (25%, 50%, 75%, 90%). At first sight this was a surprising result because the NASA_2 dataset is more balanced than NASA_1, having 29% TP messages and 71% FP messages.




(a) Bubble plot of TP/FP per file for NASA_1. The size of the bubbles represents the number of files with a given ratio of TP/FP. For example at coordinates (1,0), that is 0 TP and 1 FP, there are a total of 192 files represented by the largest bubble. The points at (0,11) and (0,19) each represent a single file containing only true positives and no false positives.




(b) Bubble plot of TP/FP per checker for NASA_1. The largest bubble is at (1,0) with a size of 3. The most interesting here are the three rightmost bubbles, each representing one checker that produced no TP messages and 126, 209, and 342 FP messages, respectively.

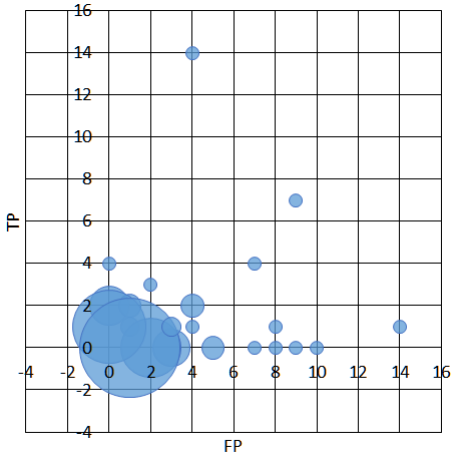
Figure 4.1: Bubble plots for NASA_1 results per file and checker

NASA_2 results were not as good as the ones for NASA_1 because, instead of having files with either only TP or only FP messages, in many files TP messages were mixed with FP messages. This can clearly be seen in Figure 4.2a: one file contains 14 TPs and 4 FPs, another contains 7 TPs and 9 FPs, 21 files contain two FPs each, and nine files contain two TPs each. The analysis per checker produced similar results; many checkers produced a mix of TP and FP results.

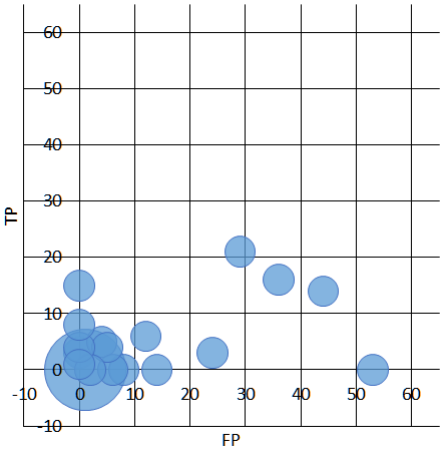
All of the rules generated from the NASA_2 data set were simple rules (i.e., had only one feature, either checker or filename) as in the following example:

checker=ANOTHER.CHECKER ==> result=TP

The results for the NASA_3 dataset are not particularly good. For both the 25% and 50% training experiments the TP recall reported was 0 while for the remaining two, training on 75% and 90%, the TP recall value increased to 0.25 and 0.35, respectively. As can be seen from Figure 4.3a, the TP/FP distribution in NASA_3's files did not lead to a good classification because TP messages are distributed across multiple files, mixed with FP messages. This observation, combined with




(a) Bubble plot for TP/FP per file for NASA_2. As with Figure 4.1a, the size of the bubbles represent the number of files per TP/FP ratio. The largest bubble is found at (1,0) and represents 58 files with 0 TP and 1 FP.



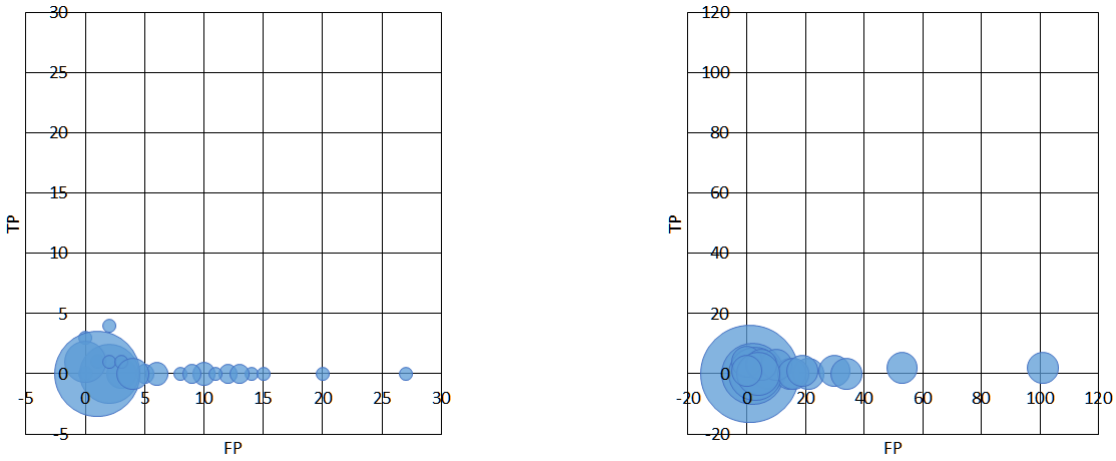

(b) Bubble plot for TP/FP per checker for NASA_2. The largest bubble is again at (1,0) with a size of 7. The right most bubble on the x axis corresponds to one checker that produced 53 FP and 0 TP.

Figure 4.2: Bubble plots for NASA_2 results per file and checker

the very small number of TP (only 5%), led to the worst classification results of the five datasets.

Similar to the NASA_3 dataset, the NASA_4 dataset is very unbalanced. Furthermore, the results obtained from NASA_4 are very comparable to the results obtained from NASA_3, that is poor TP recall of 0.16 to 0.33. Figures 4.4a and 4.4b show the true and false positive distributions. False positives per file seem to cluster quite a lot with other false positives, however this does not help the classification performance.

The NASA_5 dataset is one of the two balanced datasets experimented with in this work. Intuitively NASA_5 should provide somewhat better results than other datasets. This intuitive expectation is justified by the results: NASA_5 displays better TP recall than all other datasets with the exception of NASA_1, with a value of 0.69. Figure 4.5a shows close to “L” shaped distribution, which as previously discussed signifies source files with only TP and only FP instances.




(a) Bubble plot for TP/FP per file for NASA_3. The largest bubble is at (1,0) with a size of 44.

(b) Bubble plot for TP/FP per checker for NASA_3. The largest bubble is at (1,0) with a size of 10.

Figure 4.3: Bubble plots for NASA_3 results per file and checker

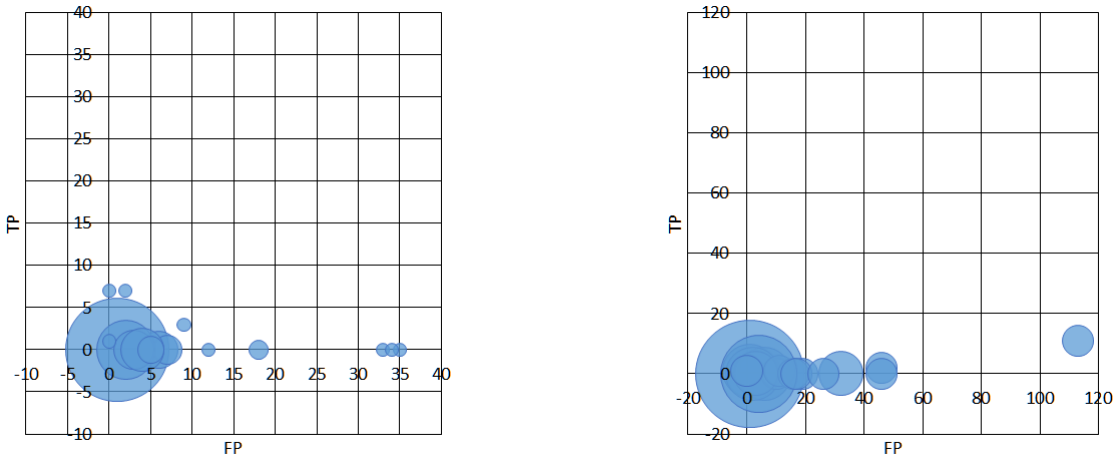
### 4.3.1 Discussion of Classification Experiments

The classification experiment presented in this thesis was motivation to explore the distribution of the analyzed true and false positive messages from real projects. This was done because the distribution of TP and FP messages can be used to explain the results. Furthermore, the way in which true and false positives are distributed throughout a codebase is interesting because based on the knowledge of how true and false positives tend to cluster one can build better classifiers and improve performance. It is desirable to know if there are many files and checkers which are associated with only one message, otherwise known as singletons. It would also be important to know if there are many files and checkers that are associated with only true positives or only false positives. Furthermore it would also be desirable to know what the population of mixed true and false positive messages per file and checker looks like.

The main observations are as follows:

- CBA works well when either only real faults and/or only false faults are clustered in files or are flagged by the same checkers. It does not work well when true and false positives are mixed together.

- By comparing the bubble plots for files and checkers in the datasets that yielded good results,




(a) Bubble plot for TP/FP per file for NASA_4. The largest bubble is at (1,0) with a size of 62.

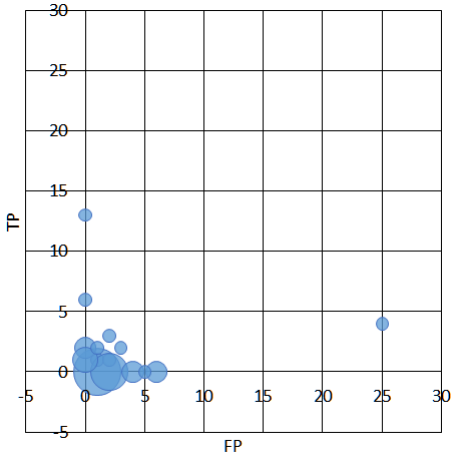
(b) Bubble plot for TP/FP per checker for NASA_4. The largest bubble is at (1,0) with a size of 12.

Figure 4.4: Bubble plots for NASA_4 results per file and checker

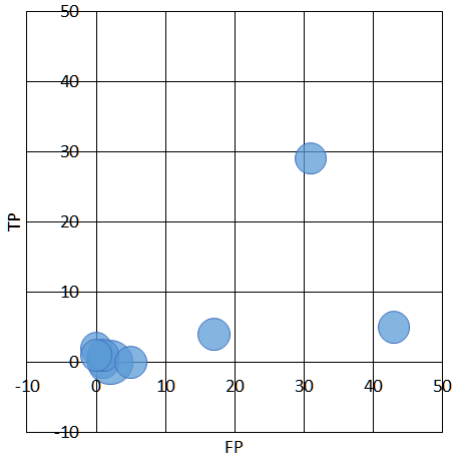
that is, NASA_1 and NASA_5, one can notice that the desirable "L" shape appears when graphing TP/FP per file yet does not appear when graphing TP/FP per checker. This does not necessarily mean that checkers are not good predictors for true and false positives.

- The "L" shape described by the Stanford team [24] does not exist for all datasets.
- There are many singletons both per file and per checker and they have a negative effect on classification performance.
- Looking at the bubble plots for both files and checkers it is observed that the majority of files and checkers exhibit 0 TP and 1 FP. That is, there are many FP singletons and near-singletons, files or checkers with, for example, 0 TP and only a few FP.

There are many singletons both per file and per checker. This does not bode well for the performance of not just the algorithm employed in this thesis but any machine learning algorithm. When only one message is flagged in a given file or by a given checker one of the features (either file or checker) is inherently void of information since it will only apply to that message. In fact when a message is a file singleton then two of the features extracted cannot provide information. Since the function feature follows the file feature both of them are useless. This trend of having




(a) Bubble plot for TP/FP per file for NASA_5. The largest bubble is at (1,0) with a size of 15. Here one can see here the desirable L shape that signifies files with only TP instances and files with only FP instances.




(b) Bubble plot for TP/FP per checker for NASA_5. The largest bubble is at (2,0) with a size of 2.

Figure 4.5: Bubble plots for NASA_5 results per file and checker

many singletons was seen for both files and checkers. However, looking at the bubble graphs one notices that the files had many more singletons (relative to their non-singleton populations) compared to checkers. Looking at the graphs one can be certain that in almost every case the largest two bubbles will be the singleton population which is to be expected given the previous observations.

The conclusion that can be drawn from these experiments is that CBA works well when the real faults (i.e., TP) and FP are clustered in several files or are flagged by the same checkers. In other words CBA performs well when true positives and false positives are found in homogeneous sets. This phenomenon resulted in the "L" shape of the scatter plot for NASA_1 shown in Figure 4.1a, which lead to TP recall of 91% using only 50% labeled messages. Contrary to expectations based on TP/FP distribution among files NASA_5 performed well compared to the other datasets. The TP recall values of 69% for the NASA_5 dataset, the second best performing of the five, show that close to "L" shape leads to acceptable performance. Thus, the data points at 17 FP/4 TP and 43 FP/5 TP in Figure 4.5b support predicting for false positives due to significantly higher number of false positives compared to true positives. The observation that the "L" shape is not necessary

for good classification performance implies that it is not a particularly generalizable concept and only applies to a specific subset of software projects.

The results for NASA_2 were significantly worse than NASA_1, even though NASA_1 had only 5% TP messages, while NASA_2 had 29% TP messages. Unlike in the case of NASA_1, the NASA_2 files had a mix of TP and FP messages, and therefore the classification was not as good, even though the percentage of TP messages was higher. As for the NASA_3 dataset, the distributions of the TP and FP messages across files and checkers were mixed, as in case of NASA_2. This combined with the low percentage of TP messages (only 5%) led to non-descriptive rules and produced very poor classification results. The NASA_4 dataset was also of mixed TP/FP distribution similar to NASA_5, however due to the more localized distribution of TPs and FPs in NASA_4 the classification performance was better.

## 4.4 Threats to Validity of Classification Experiments

This section discusses several threats to validity which may have had an effect on the work presented in this chapter. The threats are discussed in terms of construct, internal, conclusion, and external validity. This section also outlines the methods used in mitigating these threats to validity.

### 4.4.1 Construct Validity

One thread to construct validity stems from the the provenance of the data. The datasets contain only those messages which were deemed critical enough, or otherwise selected due to some other criteria, to be manually analyzed. Given the large amount of messages produced by static code analysis tool only a subset of the total number of messages per project were selected for manual analysis. The exact selection criteria used to choose messages for manual analysis were not available; however, it is known that more critical messages were more likely to be selected. This means that the whole picture with regard to TP/FP distribution is not available here, it is only a picture based on a sample of more critical messages. That being said, this is a typical situation arising in the static code analysis process and the prevalence of false positives. Because of the large number of messages no organization has the resources required to analyze every single message.

Yet another threat to construct validity appears due to the nature of static code analysis. Since a static code analysis tool only outputs messages when the tool believes a fault has been detected, the number of true negatives and false negatives cannot be assessed. This prevents studying the distribution of all faults (per file or checker) because it is likely that the static analyzer has failed to report some faults, i.e., has false negatives. Again it should be noted that the distributions discussed are based on reported and analyzed messages.

### 4.4.2 Internal Validity

The classification labels in the datasets employed were generated by manual analysis of static report messages done by NASA analysts. This is a threat to internal validity due to the possibility of human misclassification of messages. One has to trust and assume that the analysts correctly identified the true positive and false positive determinations. However, since the datasets used in this work were extracted from static analyses of real mission critical NASA systems it is believed this threat to internal validity to be acceptably mitigated.

### 4.4.3 Conclusion Validity

One threat to conclusion validity that requires mention is the size of the datasets used. Having small datasets could impact researchers' ability to draw reliable conclusions about the data at hand. In this case the datasets are not very small. Furthermore, given that the datasets were composed of static analysis messages which were actually analyzed by a human analyst, that is, messages which were given a high priority by the analysis teams, it is believed the data in question is of good quality and the size of the datasets are appropriate.

### 4.4.4 External Validity

External validity is concerned with the generalizability of research outcomes. Using five real projects provides strong empirical evidence for the external validity. Furthermore, the results presented in this work seem to imply a more general observation than found in previous work [24].

## 4.5 Conclusion on Classification Experiments

The work presented in this chapter is based on an investigation of association rule based classification of static code analysis messages. The experimental approach emulated an ongoing analysis effort across five datasets sourced from real NASA project analyses. The approach, based on training the CBA classifier on 25%, 50%, 75%, and 90% of each dataset, allowed for an investigation of how machine learning classification performs on real software projects. In addition, it also allowed for a study of the distributions of true and false positive messages across source code files and checkers and how the distributions affect the classification performance.

Using only features directly extracted from analysis reports, the results show that classification performance using association rules is dependent on whether or not true positive and false positive messages were found clustered together or spread throughout the source code. This was the case for two out of the five datasets employed. The two datasets with more well-clustered true and false positive messages resulted in good classification performance.

Message classification based on CBA as described in this chapter was implemented as part of SCAPE (Static Code Analysis Prioritization Engine), a tool designed to aid software analysts in working with static code analysis tools' reports. This tool is currently used to support static code analysis of NASA projects at the NASA IV&V facility. The tool development work was funded by NASA IV&V under a project managed by TASC.

One avenue for further research would be to investigate classification performance when additional features, such as static code metrics, or change metrics, are included.

# References

[1] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273–286, 2007.

[2] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in *15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC'09.* IEEE, 2009, pp. 301–306.

[3] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013.

[4] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting," *Software: Practice and Experience*, 2012.

[5] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities-does experience matter?" in *International Conference on Availability, Reliability and Security, 2009. ARES'09.* IEEE, 2009, pp. 804–810.

[6] P. E. Black, "Samate and evaluating static analysis tools," *Ada User Journal*, vol. 28, no. 3, pp. 184–188, 2007.

[7] (2012, December) Cas static analysis tool study - methodology. Center for Assured Software, National Security Agency. [Online]. Available: http://samate.nist.gov/docs/CAS 2012 Static Analysis Tool Study Methodology.pdf

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

[9] (2013, September) Common weakness enumeration. [Online]. Available: https://cwe.mitre.org/

[10] (2013, July) CWE development concepts. [Online]. Available: https://cwe.mitre.org/data/graphs/699.html

[11] T. Devine, K. Goseva-Popstojanova, S. Krishnan, and R. Lutz, "Assessment and cross-product prediction of software product line quality: accounting for reuse across products, over multiple releases," *Automated Software Engineering*, pp. 1–50, 2014.

[12] G. Díaz and J. R. Bermejo, "Static analysis of source code security: assessment of tools against samate tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, 2013.

[13] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, 2008.

[14] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, pp. 1–1.

[15] K. Erno, "Sticking to the facts II: Scientific study of static analysis tools," in *Presentation at the SATE IV Workshop*. Center for Assured Software, National Security Agency, March 2012.

[16] (2015) Findbugs. [Online]. Available: http://findbugs.sourceforge.net/

[17] K. Goseva-Popstojanova and A. Perhinschi, "On the performance of static code analysis for detection of security vulnerabilities," *Submitted for publication*, 2015.

[18] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, 2009.

[19] S. Heckman and L. Williams, "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*. ACM, 2008, pp. 41–50.

[20] ——, "A model building process for identifying actionable static analysis alerts," in *International Conference on Software Testing Verification and Validation, 2009. ICST'09.* IEEE, 2009, pp. 161–170.

[21] T. Hofer, "Evaluating static source code analysis tools," Master's thesis, École Polytechnique Fédérale de Lausanne, April 2010.

[22] M. Johns, M. Jodeit, W. Koeppl, and M. Wimmer, "Scanstud: Evaluating static analysis tools," in *OWASP Europe*, 2008.

[23] (2013) Juliet test suite. [Online]. Available: http://samate.nist.gov/SRD/testsuite.php

[24] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 83–93.

[25] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*. AAAI, 1998.

[26] V. Okun, A. Delaitre, and P. E. Black, "Report on the third static analysis tool exposition (sate 2010)," SP-500-283, US Nat. Inst. Stand. Techn, Tech. Rep., 2011.

[27] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 55–64.

[28] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 341–350.

[29] (2013) Static analysis tool exposition. [Online]. Available: http://samate.nist.gov/SATE.html

[30] (2013) Static analysis tool exposition. [Online]. Available: http://samate.nist.gov/SATE4.html

[31] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69–82, 2006.

[32] L. M. R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill, "Towards modeling the behavior of static code analysis tools," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*. ACM, 2014, pp. 17–20.

[33] (2014) Weka jcba implementation. [Online]. Available: http://weka.sourceforge.net/doc.packages/classAssociationRules/weka/classifiers/rules/car/JCBA.html

[34] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

# Appendix A

This appendix provides the entirety of the data presented and discussed in the static code analysis tool evaluation chapter. Detailed listing of all CWEs employed in the evaluation chapter are also included.

Table A.1: List of C/C++ CWEs and the number of associated good and bad functions along with number of test cases used in the experiments

| CWE (ID) Description | # of good functions | # of bad functions | # of test cases |
|---|---|---|---|
| (78) OS Command Injection | 5,770 | 4,410 | 4,410 |
| (122) Heap Based Buffer Overflow | 8,407 | 5,665 | 5,665 |
| (134) Uncontrolled Format String | 7,440 | 2,820 | 2,820 |
| (197) Numeric Truncation Error | 1,116 | 846 | 846 |
| (242) Use of Inherently Dangerous Function | 32 | 19 | 19 |
| (367) TOC TOU | 64 | 38 | 38 |
| (391) Unchecked Error Condition | 96 | 57 | 57 |
| (401) Memory Leak | 4,434 | 1,684 | 1,684 |
| (415) Double Free | 2,154 | 799 | 799 |
| (416) Use After Free | 1,256 | 410 | 410 |
| (457) Use of Uninitialized Variable | 2,940 | 924 | 924 |
| (467) Use of sizeof on Pointer Type | 186 | 141 | 141 |
| (468) Incorrect Pointer Scaling | 65 | 39 | 39 |
| (476) NULL Pointer Dereference | 733 | 270 | 270 |
| (478) Missing Default Case in Switch | 32 | 19 | 19 |
| (480) Use of Incorrect Operator | 32 | 19 | 19 |
| (482) Comparing Instead of Assigning | 32 | 19 | 19 |
| (561) Dead Code | 2 | 2 | 2 |
| (562) Return of Stack Variable Address | 3 | 3 | 3 |
| (563) Unused Variable | 878 | 396 | 396 |
| (590) Free Memory Not on Heap | 3,534 | 2,679 | 2,679 |
| (835) Infinite Loop | 7 | 6 | 6 |
| Total | 39,213 | 21,265 | 21,265 |

Table A.2: List of Java CWEs and the number of associated good and bad functions along with number of test cases used in the experiments

| CWE (ID) Description | # of good functions | # of bad functions | # of test cases |
|---|---|---|---|
| (80) XSS | 636 | 456 | 456 |
| (113) HTTP Response Splitting | 5,088 | 1,824 | 1,824 |
| (129) Improper Validation of Array Index | 6,148 | 2,204 | 2,204 |
| (190) Integer Overflow | 7,314 | 2,622 | 2,622 |
| (259) Hard Coded Password | 159 | 114 | 114 |
| (330) Insufficiently Random Values | 31 | 18 | 18 |
| (382) Use of System Exit | 31 | 18 | 18 |
| (396) Catch Generic Exception | 62 | 36 | 36 |
| (404) Improper Resource Shutdown | 3 | 3 | 3 |
| (476) NULL Pointer Dereference | 422 | 151 | 151 |
| (486) Compare Classes by Name | 31 | 18 | 18 |
| (489) Leftover Debug Code | 31 | 18 | 18 |
| (497) Exposure of System Data | 31 | 18 | 18 |
| (500) Public Static Field Not Final | 1 | 1 | 1 |
| (568) Finalize Without Super | 1 | 1 | 1 |
| (570) Expression Always FALSE | 6 | 6 | 6 |
| (571) Expression Always TRUE | 6 | 6 | 6 |
| (580) Clone Without Super | 1 | 1 | 1 |
| (581) Object Model Violation | 1 | 1 | 1 |
| Total | 20,003 | 7,516 | 7,516 |

Table A.3: Accuracy for C/C++

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (78) OS Command Injection | 29.27% | 57.32% | 56.88% |
| (122) Heap-based Buffer Overflow | 61.31% | 68.84% | 88.39% |
| (134) Uncontrolled Format String | 27.49% | 69.76% | 59.17% |
| (197) Numeric Truncation Error | 56.88% | 56.88% | 56.88% |
| (242) Use of Inherently Dangerous Function | 41.18% | 62.75% | 41.18% |
| (367) Time-of-check Time-of-use (TOCTOU) Race Condition | 68.63% | 62.75% | 100.00% |
| (391) Unchecked Error Condition | 62.75% | 62.75% | 62.75% |
| (401) Improper Release of Memory Before Removing Last Reference ('Memory Leak') | 59.12% | 83.16% | 94.87% |
| (415) Double Free | 65.02% | 72.40% | 92.56% |
| (416) Use After Free | 66.81% | 86.61% | 90.01% |
| (457) Use of Uninitialized Variable | 76.76% | 80.69% | 89.29% |
| (467) Use of sizeof() on a Pointer Type | 56.88% | 65.14% | 56.88% |
| (468) Incorrect Pointer Scaling | 62.50% | 51.92% | 62.50% |
| (476) NULL Pointer Dereference | 67.00% | 80.06% | 90.43% |
| (478) Missing Default Case in Switch Statement | 62.75% | 62.75% | 62.75% |
| (480) Use of Incorrect Operator | 62.75% | 62.75% | 62.75% |
| (482) Comparing instead of Assigning | 62.75% | 62.75% | 62.75% |
| (561) Dead Code | 75.00% | 75.00% | 75.00% |
| (562) Return of Stack Variable Address | 50.00% | 50.00% | 66.67% |
| (563) Unused Variable | 74.25% | 69.94% | 69.40% |
| (590) Free of Memory not on the Heap | 56.88% | 65.83% | 86.96% |
| (835) Loop with Unreachable Exit Condition ('Infinite Loop') | 53.85% | 53.85% | 53.85% |

Table A.4: Recall for C/C++

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (78) OS Command Injection | 53.51% | 3.51% | 0.00% |
| (122) Heap-based Buffer Overflow | 11.70% | 37.83% | 72.84% |
| (134) Uncontrolled Format String | 29.79% | 38.17% | 78.72% |
| (197) Numeric Truncation Error | 0.00% | 0.00% | 0.00% |
| (242) Use of Inherently Dangerous Function | 100.00% | 0.00% | 100.00% |
| (367) Time-of-check Time-of-use (TOCTOU) Race Condition | 50.00% | 0.00% | 100.00% |
| (391) Unchecked Error Condition | 0.00% | 0.00% | 0.00% |
| (401) Improper Release of Memory Before Removing Last Reference ('Memory Leak') | 48.93% | 65.91% | 81.83% |
| (415) Double Free | 17.52% | 16.77% | 73.25% |
| (416) Use After Free | 23.17% | 87.07% | 59.53% |
| (457) Use of Uninitialized Variable | 15.58% | 36.58% | 57.21% |
| (467) Use of sizeof() on a Pointer Type | 0.00% | 87.23% | 0.00% |
| (468) Incorrect Pointer Scaling | 0.00% | 48.72% | 0.00% |
| (476) NULL Pointer Dereference | 35.19% | 64.44% | 65.33% |
| (478) Missing Default Case in Switch Statement | 0.00% | 0.00% | 0.00% |
| (480) Use of Incorrect Operator | 0.00% | 0.00% | 0.00% |
| (482) Comparing instead of Assigning | 0.00% | 0.00% | 0.00% |
| (561) Dead Code | 50.00% | 50.00% | 50.00% |
| (562) Return of Stack Variable Address | 0.00% | 0.00% | 33.33% |
| (563) Unused Variable | 32.83% | 3.28% | 13.84% |
| (590) Free of Memory not on the Heap | 0.00% | 41.66% | 69.76% |
| (835) Loop with Unreachable Exit Condition ('Infinite Loop') | 0.00% | 0.00% | 0.00% |

Table A.5: Probability of False Alarm for C/C++

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (78) OS Command Injection | 89.25% | 1.56% | 0.00% |
| (122) Heap-based Buffer Overflow | 5.26% | 10.21% | 1.60% |
| (134) Uncontrolled Format String | 73.39% | 21.37% | 48.36% |
| (197) Numeric Truncation Error | 0.00% | 0.00% | 0.00% |
| (242) Use of Inherently Dangerous Function | 93.75% | 0.00% | 93.75% |
| (367) Time-of-check Time-of-use (TOCTOU) Race Condition | 20.31% | 0.00% | 0.00% |
| (391) Unchecked Error Condition | 0.00% | 0.00% | 0.00% |
| (401) Improper Release of Memory Before Removing Last Reference ('Memory Leak') | 37.01% | 10.28% | 0.00% |
| (415) Double Free | 17.36% | 6.96% | 0.00% |
| (416) Use After Free | 18.95% | 13.54% | 0.00% |
| (457) Use of Uninitialized Variable | 4.01% | 5.44% | 0.63% |
| (467) Use of sizeof() on a Pointer Type | 0.00% | 51.61% | 0.00% |
| (468) Incorrect Pointer Scaling | 0.00% | 46.15% | 0.00% |
| (476) NULL Pointer Dereference | 21.28% | 14.19% | 0.00% |
| (478) Missing Default Case in Switch Statement | 0.00% | 0.00% | 0.00% |
| (480) Use of Incorrect Operator | 0.00% | 0.00% | 0.00% |
| (482) Comparing instead of Assigning | 0.00% | 0.00% | 0.00% |
| (561) Dead Code | 0.00% | 0.00% | 0.00% |
| (562) Return of Stack Variable Address | 0.00% | 0.00% | 0.00% |
| (563) Unused Variable | 7.06% | 0.00% | 3.66% |
| (590) Free of Memory not on the Heap | 0.00% | 15.85% | 0.00% |
| (835) Loop with Unreachable Exit Condition ('Infinite Loop') | 0.00% | 0.00% | 0.00% |

Table A.6: Balance for C/C++

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (78) OS Command Injection | 28.84% | 31.77% | 29.29% |
| (122) Heap-based Buffer Overflow | 37.45% | 55.45% | 80.76% |
| (134) Uncontrolled Format String | 28.18% | 53.74% | 62.64% |
| (197) Numeric Truncation Error | 29.29% | 29.29% | 29.29% |
| (242) Use of Inherently Dangerous Function | 33.71% | 29.29% | 33.71% |
| (367) Time-of-check Time-of-use (TOCTOU) Race Condition | 61.84% | 29.29% | 100.00% |
| (391) Unchecked Error Condition | 29.29% | 29.29% | 29.29% |
| (401) Improper Release of Memory Before Removing Last Reference ('Memory Leak') | 55.40% | 74.82% | 87.15% |
| (415) Double Free | 40.40% | 40.94% | 81.09% |
| (416) Use After Free | 44.05% | 86.77% | 71.38% |
| (457) Use of Uninitialized Variable | 40.24% | 54.99% | 69.74% |
| (467) Use of sizeof() on a Pointer Type | 29.29% | 62.40% | 29.29% |
| (468) Incorrect Pointer Scaling | 29.29% | 51.21% | 29.29% |
| (476) NULL Pointer Dereference | 51.76% | 72.93% | 75.49% |
| (478) Missing Default Case in Switch Statement | 29.29% | 29.29% | 29.29% |
| (480) Use of Incorrect Operator | 29.29% | 29.29% | 29.29% |
| (482) Comparing instead of Assigning | 29.29% | 29.29% | 29.29% |
| (561) Dead Code | 64.64% | 64.64% | 64.64% |
| (562) Return of Stack Variable Address | 29.29% | 29.29% | 52.86% |
| (563) Unused Variable | 52.24% | 31.61% | 39.02% |
| (590) Free of Memory not on the Heap | 29.29% | 57.25% | 78.62% |
| (835) Loop with Unreachable Exit Condition ('Infinite Loop') | 29.29% | 29.29% | 29.29% |

Table A.7: G-score for C/C++

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (78) OS Command Injection | 17.90% | 6.79% | 0.00% |
| (122) Heap-based Buffer Overflow | 20.83% | 53.24% | 83.71% |
| (134) Uncontrolled Format String | 28.11% | 51.39% | 62.37% |
| (197) Numeric Truncation Error | 0.00% | 0.00% | 0.00% |
| (242) Use of Inherently Dangerous Function | 11.76% | 0.00% | 11.76% |
| (367) Time-of-check Time-of-use (TOCTOU) Race Condition | 61.45% | 0.00% | 100.00% |
| (391) Unchecked Error Condition | 0.00% | 0.00% | 0.00% |
| (401) Improper Release of Memory Before Removing Last Reference ('Memory Leak') | 55.08% | 76.00% | 90.01% |
| (415) Double Free | 28.91% | 28.42% | 84.56% |
| (416) Use After Free | 36.04% | 86.77% | 74.63% |
| (457) Use of Uninitialized Variable | 26.82% | 52.75% | 72.61% |
| (467) Use of sizeof() on a Pointer Type | 0.00% | 62.25% | 0.00% |
| (468) Incorrect Pointer Scaling | 0.00% | 51.15% | 0.00% |
| (476) NULL Pointer Dereference | 48.63% | 73.61% | 79.03% |
| (478) Missing Default Case in Switch Statement | 0.00% | 0.00% | 0.00% |
| (480) Use of Incorrect Operator | 0.00% | 0.00% | 0.00% |
| (482) Comparing instead of Assigning | 0.00% | 0.00% | 0.00% |
| (561) Dead Code | 66.67% | 66.67% | 66.67% |
| (562) Return of Stack Variable Address | 0.00% | 0.00% | 50.00% |
| (563) Unused Variable | 48.52% | 6.36% | 24.20% |
| (590) Free of Memory not on the Heap | 0.00% | 55.73% | 82.19% |
| (835) Loop with Unreachable Exit Condition ('Infinite Loop') | 0.00% | 0.00% | 0.00% |

Table A.8: Accuracy for Java

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (80) XSS | 59.25% | 68.13% | 58.79% |
| (113) HTTP Response Splitting | 60.69% | 61.98% | 72.77% |
| (129) Improper Validation of Array Index | 73.61% | 71.74% | 73.24% |
| (190) Integer Overflow | 73.61% | 64.63% | 73.24% |
| (259) Hard Coded Password | 58.97% | 58.97% | 61.90% |
| (330) Insufficiently Random Values | 40.82% | 40.82% | 63.27% |
| (382) Use of System Exit | 40.82% | 63.27% | 71.43% |
| (396) Catch Generic Exception | 63.27% | 0.00% | 52.04% |
| (404) Improper Resource Shutdown | 66.67% | 50.00% | 66.67% |
| (476) NULL Pointer Dereference | 59.51% | 73.65% | 84.60% |
| (486) Compare Classes by Name | 63.27% | 63.27% | 63.27% |
| (489) Leftover Debug Code | 63.27% | 63.27% | 63.27% |
| (497) Exposure of System Data | 40.82% | 40.82% | 63.27% |
| (500) Public Static Field Not Final | 100.00% | 50.00% | 100.00% |
| (568) Finalize Without Super | 50.00% | 100.00% | 100.00% |
| (570) Expression Always FALSE | 75.00% | 50.00% | 58.33% |
| (571) Expression Always TRUE | 75.00% | 50.00% | 58.33% |
| (580) Clone Without Super | 100.00% | 50.00% | 100.00% |
| (581) Object Model Violation | 100.00% | 100.00% | 100.00% |

Table A.9: Recall for Java

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (80) XSS | 55.26% | 69.08% | 4.39% |
| (113) HTTP Response Splitting | 36.79% | 34.54% | 4.39% |
| (129) Improper Validation of Array Index | 0.00% | 9.53% | 0.00% |
| (190) Integer Overflow | 0.00% | 26.70% | 0.00% |
| (259) Hard Coded Password | 2.63% | 30.70% | 8.77% |
| (330) Insufficiently Random Values | 100.00% | 100.00% | 0.00% |
| (382) Use of System Exit | 100.00% | 0.00% | 100.00% |
| (396) Catch Generic Exception | 0.00% | 0.00% | 50.00% |
| (404) Improper Resource Shutdown | 100.00% | 0.00% | 33.33% |
| (476) NULL Pointer Dereference | 37.75% | 0.00% | 47.68% |
| (486) Compare Classes by Name | 0.00% | 0.00% | 0.00% |
| (489) Leftover Debug Code | 0.00% | 0.00% | 0.00% |
| (497) Exposure of System Data | 100.00% | 100.00% | 0.00% |
| (500) Public Static Field Not Final | 100.00% | 0.00% | 100.00% |
| (568) Finalize Without Super | 0.00% | 100.00% | 100.00% |
| (570) Expression Always FALSE | 50.00% | 0.00% | 16.67% |
| (571) Expression Always TRUE | 50.00% | 0.00% | 16.67% |
| (580) Clone Without Super | 100.00% | 0.00% | 100.00% |
| (581) Object Model Violation | 100.00% | 100.00% | 100.00% |

Table A.10: Probability of False Alarm for Java

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (80) XSS | 37.89% | 32.55% | 2.20% |
| (113) HTTP Response Splitting | 30.74% | 28.18% | 2.24% |
| (129) Improper Validation of Array Index | 0.00% | 5.95% | 0.00% |
| (190) Integer Overflow | 0.00% | 21.77% | 0.00% |
| (259) Hard Coded Password | 0.63% | 20.75% | 0.00% |
| (330) Insufficiently Random Values | 93.55% | 93.55% | 0.00% |
| (382) Use of System Exit | 93.55% | 0.00% | 45.16% |
| (396) Catch Generic Exception | 0.00% | 100.00% | 46.77% |
| (404) Improper Resource Shutdown | 66.67% | 0.00% | 0.00% |
| (476) NULL Pointer Dereference | 32.70% | 0.00% | 1.93% |
| (486) Compare Classes by Name | 0.00% | 0.00% | 0.00% |
| (489) Leftover Debug Code | 0.00% | 0.00% | 0.00% |
| (497) Exposure of System Data | 93.55% | 93.55% | 0.00% |
| (500) Public Static Field Not Final | 0.00% | 0.00% | 0.00% |
| (568) Finalize Without Super | 0.00% | 0.00% | 0.00% |
| (570) Expression Always FALSE | 0.00% | 0.00% | 0.00% |
| (571) Expression Always TRUE | 0.00% | 0.00% | 0.00% |
| (580) Clone Without Super | 0.00% | 0.00% | 0.00% |
| (581) Object Model Violation | 0.00% | 0.00% | 0.00% |

Table A.11: Balance for Java

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (80) XSS | 58.54% | 68.26% | 32.37% |
| (113) HTTP Response Splitting | 50.30% | 49.60% | 32.37% |
| (129) Improper Validation of Array Index | 29.29% | 35.89% | 29.29% |
| (190) Integer Overflow | 29.29% | 45.93% | 29.29% |
| (259) Hard Coded Password | 31.15% | 48.85% | 35.49% |
| (330) Insufficiently Random Values | 33.85% | 33.85% | 29.29% |
| (382) Use of System Exit | 33.85% | 29.29% | 68.07% |
| (396) Catch Generic Exception | 29.29% | 0.00% | 51.59% |
| (404) Improper Resource Shutdown | 52.86% | 29.29% | 52.86% |
| (476) NULL Pointer Dereference | 50.28% | 29.29% | 62.98% |
| (486) Compare Classes by Name | 29.29% | 29.29% | 29.29% |
| (489) Leftover Debug Code | 29.29% | 29.29% | 29.29% |
| (497) Exposure of System Data | 33.85% | 33.85% | 29.29% |
| (500) Public Static Field Not Final | 100.00% | 29.29% | 100.00% |
| (568) Finalize Without Super | 29.29% | 100.00% | 100.00% |
| (570) Expression Always FALSE | 64.64% | 29.29% | 41.07% |
| (571) Expression Always TRUE | 64.64% | 29.29% | 41.07% |
| (580) Clone Without Super | 100.00% | 29.29% | 100.00% |
| (581) Object Model Violation | 100.00% | 100.00% | 100.00% |

Table A.12: G-score for Java

| CWE (ID) Description | Tool A | Tool B | Tool C |
|---|---|---|---|
| (80) XSS | 58.49% | 68.26% | 8.40% |
| (113) HTTP Response Splitting | 48.05% | 46.65% | 8.40% |
| (129) Improper Validation of Array Index | 0.00% | 17.30% | 0.00% |
| (190) Integer Overflow | 0.00% | 39.81% | 0.00% |
| (259) Hard Coded Password | 5.13% | 44.26% | 16.13% |
| (330) Insufficiently Random Values | 12.12% | 12.12% | 0.00% |
| (382) Use of System Exit | 12.12% | 0.00% | 70.83% |
| (396) Catch Generic Exception | 0.00% | 0.00% | 51.56% |
| (404) Improper Resource Shutdown | 50.00% | 0.00% | 50.00% |
| (476) NULL Pointer Dereference | 48.37% | 0.00% | 64.17% |
| (486) Compare Classes by Name | 0.00% | 0.00% | 0.00% |
| (489) Leftover Debug Code | 0.00% | 0.00% | 0.00% |
| (497) Exposure of System Data | 12.12% | 12.12% | 0.00% |
| (500) Public Static Field Not Final | 100.00% | 0.00% | 100.00% |
| (568) Finalize Without Super | 0.00% | 100.00% | 100.00% |
| (570) Expression Always FALSE | 66.67% | 0.00% | 28.57% |
| (571) Expression Always TRUE | 66.67% | 0.00% | 28.57% |
| (580) Clone Without Super | 100.00% | 0.00% | 100.00% |
| (581) Object Model Violation | 100.00% | 100.00% | 100.00% |

Table A.13: Recall for categories with C/C++

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 398 | CWE 478 | 0.000 | 0.000 | 0.000 |
| | CWE 561 | 0.500 | 0.500 | 0.500 |
| | CWE 562 | 0.000 | 0.000 | 0.333 |
| | CWE 563 | 0.328 | 0.033 | 0.138 |
| | Overall | 0.312 | 0.033 | 0.135 |
| CWE 399 | CWE 401 | 0.489 | 0.659 | 0.818 |
| | CWE 415 | 0.175 | 0.168 | 0.733 |
| | CWE 416 | 0.232 | 0.871 | 0.595 |
| | CWE 590 | 0.000 | 0.417 | 0.698 |
| | Overall | 0.190 | 0.488 | 0.731 |
| CWE 465 | CWE 467 | 0.000 | 0.872 | 0.000 |
| | CWE468 | 0.000 | 0.487 | 0.000 |
| | CWE476 | 0.352 | 0.644 | 0.653 |
| | Overall | 0.211 | 0.702 | 0.363 |
| CWE 569 | CWE 480 | 0.000 | 0.000 | 0.000 |
| | CWE 482 | 0.000 | 0.000 | 0.000 |
| | Overall | 0.000 | 0.000 | 0.000 |
| CWE 633 | CWE 122 | 0.117 | 0.378 | 0.728 |
| | CWE 134 | 0.298 | 0.382 | 0.787 |
| | Overall | 0.177 | 0.379 | 0.743 |

Table A.14: Probability of False Alarm for categories with C/C++

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 398 | CWE 478 | 0.000 | 0.000 | 0.000 |
| | CWE 561 | 0.000 | 0.000 | 0.000 |
| | CWE 562 | 0.000 | 0.000 | 0.000 |
| | CWE 563 | 0.071 | 0.000 | 0.037 |
| | Overall | 0.068 | 0.000 | 0.035 |
| CWE 399 | CWE 401 | 0.370 | 0.103 | 0.000 |
| | CWE 415 | 0.174 | 0.070 | 0.000 |
| | CWE 416 | 0.189 | 0.135 | 0.000 |
| | CWE 590 | 0.000 | 0.158 | 0.000 |
| | Overall | 0.198 | 0.117 | 0.000 |
| CWE 465 | CWE 467 | 0.000 | 0.516 | 0.000 |
| | CWE 468 | 0.000 | 0.462 | 0.000 |
| | CWE 476 | 0.213 | 0.142 | 0.000 |
| | Overall | 0.159 | 0.234 | 0.000 |
| CWE 569 | CWE 480 | 0.000 | 0.000 | 0.000 |
| | CWE 482 | 0.000 | 0.000 | 0.000 |
| | Overall | 0.000 | 0.000 | 0.000 |
| CWE 633 | CWE 122 | 0.053 | 0.102 | 0.016 |
| | CWE 134 | 0.734 | 0.214 | 0.484 |
| | Overall | 0.372 | 0.165 | 0.179 |

Table A.15: G-Score for categories with C/C++

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 398 | CWE 478 | 0.000 | 0.000 | 0.000 |
| | CWE 561 | 0.667 | 0.667 | 0.667 |
| | CWE 562 | 0.000 | 0.000 | 0.500 |
| | CWE 563 | 0.485 | 0.064 | 0.242 |
| | Overall | 0.467 | 0.065 | 0.236 |
| CWE 399 | CWE 401 | 0.551 | 0.760 | 0.900 |
| | CWE 415 | 0.289 | 0.284 | 0.846 |
| | CWE 416 | 0.360 | 0.868 | 0.746 |
| | CWE 590 | 0.000 | 0.557 | 0.822 |
| | Overall | 0.307 | 0.628 | 0.845 |
| CWE 465 | CWE 467 | 0.000 | 0.622 | 0.000 |
| | CWE468 | 0.000 | 0.512 | 0.000 |
| | CWE 476 | 0.486 | 0.736 | 0.790 |
| | Overall | 0.338 | 0.733 | 0.533 |
| CWE 569 | CWE 480 | 0.000 | 0.000 | 0.000 |
| | CWE 482 | 0.000 | 0.000 | 0.000 |
| | Overall | 0.000 | 0.000 | 0.000 |
| CWE 633 | CWE 122 | 0.208 | 0.532 | 0.837 |
| | CWE 134 | 0.281 | 0.514 | 0.624 |
| | Overall | 0.276 | 0.522 | 0.780 |

Table A.16: Recall for categories with Java

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 189 | CWE 129 | 0.000 | 0.095 | 0.000 |
| | CWE 190 | 0.000 | 0.267 | 0.000 |
| | Overall | 0.000 | 0.189 | 0.000 |
| CWE 254 | CWE 259 | 0.026 | 0.307 | 0.088 |
| | CWE 330 | 1.000 | 1.000 | 0.000 |
| | Overall | 0.159 | 0.402 | 0.076 |
| CWE 399 | CWE 404 | 1.000 | 0.000 | 0.333 |
| | CWE 568 | 0.000 | 1.000 | 1.000 |
| | Overall | 0.750 | 0.250 | 0.500 |
| CWE 422 | CWE 80 | 0.553 | 0.691 | 0.044 |
| | CWE 113 | 0.368 | 0.345 | 0.044 |
| | Overall | 0.405 | 0.414 | 0.044 |
| CWE 569 | CWE 570 | 0.500 | 0.000 | 0.167 |
| | CWE 571 | 0.500 | 0.000 | 0.167 |
| | Overall | 0.500 | 0.000 | 0.167 |

Table A.17: Probability of False Alarm for categories with Java

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 189 | CWE 129 | 0.000 | 0.060 | 0.000 |
| | CWE 190 | 0.000 | 0.218 | 0.000 |
| | Overall | 0.000 | 0.145 | 0.000 |
| CWE 254 | CWE 259 | 0.006 | 0.208 | 0.000 |
| | CWE 330 | 0.935 | 0.935 | 0.000 |
| | Overall | 0.158 | 0.326 | 0.000 |
| CWE 399 | CWE 404 | 0.667 | 0.000 | 0.000 |
| | CWE 568 | 0.000 | 0.000 | 0.000 |
| | Overall | 0.500 | 0.000 | 0.000 |
| CWE 422 | CWE 80 | 0.379 | 0.325 | 0.022 |
| | CWE 113 | 0.307 | 0.282 | 0.022 |
| | Overall | 0.315 | 0.287 | 0.022 |
| CWE 569 | CWE 570 | 0.000 | 0.000 | 0.000 |
| | CWE 571 | 0.000 | 0.000 | 0.000 |
| | Overall | 0.000 | 0.000 | 0.000 |

Table A.18: G-Score for categories with Java

| Category | CWE ID | Tool A | Tool B | Tool C |
|---|---|---|---|---|
| CWE 189 | CWE 129 | 0.000 | 0.173 | 0.000 |
| | CWE 190 | 0.000 | 0.398 | 0.000 |
| | Overall | 0.000 | 0.309 | 0.000 |
| CWE 254 | CWE 259 | 0.051 | 0.443 | 0.161 |
| | CWE 330 | 0.121 | 0.121 | 0.000 |
| | Overall | 0.268 | 0.503 | 0.141 |
| CWE 399 | CWE 404 | 0.500 | 0.000 | 0.500 |
| | CWE 568 | 0.000 | 1.000 | 1.000 |
| | Overall | 0.600 | 0.400 | 0.667 |
| CWE 442 | CWE 80 | 0.585 | 0.683 | 0.084 |
| | CWE 113 | 0.481 | 0.466 | 0.084 |
| | Overall | 0.509 | 0.524 | 0.084 |
| CWE 569 | CWE 570 | 0.667 | 0.000 | 0.286 |
| | CWE 571 | 0.667 | 0.000 | 0.286 |
| | Overall | 0.667 | 0.000 | 0.286 |