USING MACHINE LEARNING TECHNIQUES TO IMPROVE STATIC CODE ANALYSIS TOOLS USEFULNESS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Enas A. Alikhashashneh

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

To My Parents, My Husband, and My Kids.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. James H. Hill, for his guidance, advice, and knowledge, without whom, this work would not have been possible. Another thanks goes out to Dr. Rajeev R. Raje, Dr. Mohammad Al Hasan, and Dr. Mihran Tuceryan for being on my thesis defense committee and for their assistance in answering various questions I had. Finally, I want to thank my family for their support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

XML	eXtensible Markup Language
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
NIST	National Institute of Standards and Technology
FN	False Negative
FP	False Positive
TP	True Positive
SCA	Static Code Analysis
DA	Domain Adaptation
SCATE	Test Suite Code Analysis Tool Evaluator
SCATWR	Static Code Analysis Tool's Warnings Ranking
SCATWC	Static Code Analysis Tool's Warnings Classification
ARFF	Attribute-Relation File Format
ML	Machine Learning
DM	Data Mining
LOC	Lines Of Code
BLOC	Blank Lines of Code
CLOC	Lines with Comments
SMOTE	Synthetic Minority Over-sampling Technique
KNN	K-Nearest Neighbor
SVM	Support Vector Machine
RF	Random Forest
RIPPER	Repeated Incremental Pruning to Produce Error Reduction
CFS	Correlation-based Feature Selection

ABSTRACT

Alikhashashneh, Enas A. Ph.D., Purdue University, August 2019. Using Machine Learning Techniques to Improve Static Code Analysis Tools Usefulness. Major Professor: James H. Hill.

This dissertation proposes an approach to reduce the cost of manual inspections for as large a number of false positive warnings that are being reported by Static Code Analysis (SCA) tools as much as possible using Machine Learning (ML) techniques. The proposed approach neither assume to use the particular SCA tools nor depends on the specific programming language used to write the target source code or the application. To reduce the number of false positive warnings we first evaluated a number of SCA tools in terms of software engineering metrics using a highlighted synthetic source code named the Juliet test suite. From this evaluation, we concluded that the SCA tools report plenty of false positive warnings that need a manual inspection. Then we generated a number of datasets from the source code that forced the SCA tool to generate either true positive, false positive, or false negative warnings. The datasets, then, were used to train four of ML classifiers in order to classify the collected warnings from the synthetic source code. From the experimental results of the ML classifiers, we observed that the classifier that built using the Random Forests (RF) technique outperformed the rest of the classifiers. Lastly, using this classifier and an instance-based transfer learning technique, we ranked a number of warnings that were aggregated from various open-source software projects. The experimental results show that the proposed approach to reduce the cost of the manual inspection of the false positive warnings outperformed the random ranking algorithm and was highly correlated with the ranked list that the optimal ranking algorithm generated.

1. INTRODUCTION

The most critical goal of the software engineering today is how to build secure software that continues functioning properly under malicious attack [1]. There are several of procedures that software developers use to build secure software, such as designing software to be secure, testing software for security issues, educating themselves by leveraging best practices of software engineering, detecting common software defects and threats such as buffer overflows and SQL injection early in the software life cycle, and minimizing potential security holes in the software by reducing its complexity and extensibility.

Additionally, there are several practices that can be applied to various software artifacts to increase the security and enhance quality. For example, abuse cases, security requirements, and risk analysis can be applied to software requirements and use cases. Further, enforcing different code analysis approaches to the software source code and/or to the compiled versions of the source code can help uncover possible security vulnerabilities. In general, the code analysis approaches can be divided into two main groups: Static Code Analysis (SCA) and Dynamic Program Analysis (DPA) [2].

DPA has proven to be an effective technique for finding potential security defects in the source code by analyzing the proprieties of a program while it is executing [3] [4]. Moreover, the DPA technique relies on program instrumentation to modify the original program source code for the purpose of gathering traces and collecting an efficient collection of run-time information to find and understand the problematic code [5]. However, DPA suffers from the overhead of program execution. Likewise, DPA can only find defects in the part of the code that is actually executed. Moreover, relying on DPA to detect possible flaws in software source code can be costly, as flaws are reported late in the software development life cycle.

Contrary to DPA, SCA examines software source code or the binary code without execution to identify potential defects as a warning message [6]. Warnings generated by SCA tools can be categorized into three main groups: false positive warnings, which incorrectly report potential defects in the source code; true positive warnings, which correctly report potential defects in the source code; and false negative warnings, which do not generate messages for buggy code [7].

As SCA does not need to run the software code and check the output, the security vulnerabilities can be detected early when it is still inexpensive to fix them. The tools that automated SCA process inspects the source code using different techniques, such as bug patterns, control-flow analysis, data-flow analysis, or lexical analysis instead of relying on input stimuli; therefore, SCA tools can be generalized for all possible program behaviors, not just to the current environment [8] [9] [10] [11]. In general, SCA tools' rules are written in ways to assist the software developers identify common software errors that the compiler cannot find, such as Common Weakness Enumeration (CWE) and SEI CERT coding rules violations.

Furthermore, some SCA tools operate on the source code, while others check the intermediate code and libraries created. Moreover, different SCA tools support different programming languages. For example, FindBugs, which is an open-source SCA tool that supports Java [12]; Pylint, which is an open-source SCA tool widely used by the Python community to reveal potential defects in the python source code [13]; and Klocwork, which is a commercial SCA tool that analyzes C/C++, C#, and Java programming languages [14] [15].

Therefore, developers may choose from a mix of open-source and commercial SCA tools, as different tools may produce different results [16]. Because software developers and testers have many SCA tools to choose from, a common challenge they face is identifying which tool to use against their code base. As mentioned above, different SCA tools have unique strengths, weaknesses, and performance characteristics, which we call their *quality*, in terms of being able to correctly identify potential vulnerabilities. The problem is exacerbated when multiple SCA tools claim to check the same vulnerabilities but generate different results. In this scenario, at least one of the SCA tools is generating both false positives and false negatives.

In the past, there have been several attempts to evaluate the quality of SCA tools. For example, Knudsen [17] evaluated the ability of Visual Code Grepper, FindBugs, and SonarQube to detect SQL, OS command, and LDAP injection vulnerabilities against the Java Juliet Test Suite [18]. Likewise, McLean [19] evaluated several SCA tools against widely used open-source applications, such as Apache OpenOffice (AOO) [20], PuTTY [21], NMAP [22], and Wireshark [23]. Further, Velicheti et al. [24] developed a framework for evaluating different SCA tools against the Juliet test suite for C++ and Java. We further discuss the related literature on SCA tools evaluation in Chapter 2.

Although there have been several attempts in the past to evaluate the quality of SCA tools, none of the existing studies has performed an in-depth analysis of SCA tools based on well-known software engineering metrics. For example, it is unknown how software engineering metrics such as Knots [25], Essential Complexity [25], Cyclomatic Complexity [25], Fan-Out (CountOutput) [26], and Fan-In (*i.e.*, CountInput) [26] impact an SCA tool's true positive rate, which is an SCA tool's ability to correctly label a flaw, false positive rate, and false negative rate. Likewise, it is unknown which software engineering metrics have the most impact on the true positive, false positive, and false negative rates for an SCA tool.

These are important questions that need to be answered because if we can understand how different software engineering metrics impact an SCA tool, then there is potential to assist tool developers in understanding the weak spots in their analytical capabilities. More important, we can provide guidelines for software engineers on how to write better code, so that SCA tools will generate fewer false positives and false negatives and, potentially, more true positives. We further defend our proposed framework for evaluating the SCA tools in Chapter 3.

However, like many other tools and techniques, SCA tools have some demerits, one of which is that the SCA tool cannot assist software developers in detecting when the software performs an unexpected operation. The second demerit of using SCA is that the SCA tool may generate a massive number of false positives and false negatives. In fact, Kremenek and Engler [27] observed that the false positive rates for some SCA tools range between 30%—100%. Other studies, such as [28] [29] [30] have indicated that 35%—91% of generated warnings are false positive.

One possible solution to this problem is to manually inspect all generated warnings and identify them as either false or true positive warnings. Manually inspecting generated warnings, however, is an expensive, time-consuming process that commonly leads software developers to reject using SCA tools or ignore warnings that may represent actual defects [31] [32]. Another solution is to automate post-processing of warning messages to reduce cost of manual inspection using ML techniques that classify and rank the generated warnings [33]. For example, if we assume that each warning needs 2 minutes to determine if it is a true or positive warning, and the software developer has 3,000 generated warnings, manual inspection will require 4.16 workdays. Conversely, ML techniques will need only 3–6 minutes.

Using ML techniques in the SCA field improves the usefulness of the SCA tools and encourages software developers to use them. Unfortunately, there are two main limitations to using the ML techniques. First, such techniques assume that there is a large training set that can be used to identify and analyze relationships between the features and label value. Generating of such a labeled dataset manually can be considered as a time-consuming, costly, and boring process. To solve this challenge, we extended the proposed framework in Chapter 3 to automatically compute the software engineering metrics and automatically generate a number of labeled datasets for the synthetic source code and a number of unlabeled datasets for open-source software projects. Then, by using the generated datasets, we show how we utilize the ML techniques to predict the SCA tool warnings based on the value of the software engineering metrics. We further discuss the proposed framework in Chapter 4.

The second limitation of using ML techniques is that the prediction model that is trained using the aggregated warnings from the synthetic source code can correctly classify and rank the warnings from the same or other synthetic source code. However, in some cases the warnings generated from an open-source software project can be too small for training and testing the prediction model. Thus, we cannot retrain the prediction model that we used in Chapter 4 to classify or rank the warnings in the open-source software projects. The best solution for this problem is for the developers or testers to take a trained prediction model from another open-source software project or form synthetic source code to successfully predict and rank the SCA tool warnings in their software.

Unfortunately, an applying prediction model obtained from synthetic source code and used in an open-source software project directly may decrease the prediction model performance, and this may lead to misclassifying some important and actual defects in the open-source software projects into false or fake warnings. Based on our knowledge, none of the current studies addresses this problem. To solve this challenge, in Chapter 5 we propose a framework for generating a number of unlabeled datasets for a set of open-source software projects and transfer the prediction model that we used in Chapter 4 to rank the warnings in the open-source software projects.

To reduce the number of false positive warnings, we hypothesize the following:

- Software engineering metrics values impact the true positive, false positive, and false negative rates of the SCA tools.
- ML techniques can be used along with a collection of software engineering metrics to predict if the source code will lead the SCA tool to emit either true positive, false positive, or false negative warnings.
- Reducing false positive warnings by using ML techniques to rank the warnings.

Our approach is as follows: we evaluate a set of SCA tools using Juliet test suite in terms of software engineering metrics. Next, we automatically generate a number of labeled datasets from the synthetic source code that force the SCA tool to generate false positive, true positive, and false negative warning. Then we train four of ML techniques using the labeled datasets to predict and classify the SCA tools' warnings. Following, we use the best prediction model that outperformed the rest of techniques to rank the SCA tool warnings generated using open-source software projects. Finally, we validate our work with open-source software developers by having them provide feedback on our warnings ranking list. Finally, sixty of computer science and engineering undergraduate students verify our work over their source code.

With this understanding, the contributions of this thesis are as follows:

- We evaluate SCA tools using a number of open-source software projects and a standardized test suite, such as the Juliet test suite in terms of software engineering metrics.
- We generate twelve datasets to better represent the source code snippets that forces the SCA tools to report a warning.
- We show the practicality of using ML techniques to classify the SCA tool warnings.
- We design a framework for ranking SCA tool's warnings and for eliminating false positive warnings from the SCA tool's output.

Our experimental results show that the proposed approach can reduce the amount of time developers must spend triaging warning messages generated by SCA tools. This is because developers can focus on warning messages that fall within a certain threshold, as opposed to investigating all generated warning messages.

The remainder of this dissertation is organized as follows: Chapter 2 discusses existing approaches from the literature for evaluating and mitigating false positives in SCA tools; Chapter 3 presents our framework for evaluating SCA tools in terms of source code metrics; Chapter 4 describes our approach for classifying SCA tool warnings into false positives, true positives, and false negatives; Chapter 5 presents our framework on filtering out and ranking false positives from a tool's output based on the confidence value that was computed using the prediction model; Finally, Chapter 6 concludes the thesis.

2. RELATED WORK

This chapter discusses other approaches in the literature for evaluating the static code analysis tools and for reducing the number of false positive reports from static code analysis tools.

2.1 Evaluating SCA Tools

Many studies have evaluated SCA tools using different test cases. For example, Knudsen et al. [17] tested the ability of three of the open-source SCA tools (Visual Code Grepper, FindBugs, and SonarQube) to detect SQL, OS command, and LDAP injection vulnerabilities against the Juliet Test Suite v1.2 for Java. The performance of these tools is evaluated using the OWASP Benchmark Project, which provides a system to test the performance of the SCA tools using the Youden index metric [34]. They conclude that FindBugs may be considered as the best tool in detecting LDAP injections. This work differs from ours in two ways. First, our work evaluates SCA tools in the context of different software engineering metrics; second, we evaluate the SCA tools against 91 CWEs, while Knudsen et al. evaluated the SCA tools against only three CWEs.

McLean et al. [19] compared two SCA tools, RATS and Flawfinder, and their ability to find vulnerabilities in three open-source applications. The results of their study concluded that the Flawfinder uncovered 3,189 flaws, while RATS found only 1,415 flaws. On the other hand, both SCA tools produce a large number of false positives. Last, these authors recommend that the developers analyze their source code using Flawfinder because this tool reports more valuable information to the developer than RATS. Our work differs from McLean et al. in that our work evaluates the

SCA tools in the context of software engineering metrics, and focuses on well-known weaknesses (i.e., CWEs) in the Juliet Test Suite.

Baca et al. [35] ran a single SCA tool over four commercial software systems from Ericsson. They concluded from the SCA tool outputs that the tool generated a large number of false positive warnings and a low percentage of true warnings. Only 37.5% of the false positive warnings were identified manually by the developers, a process that consumed much of the developers' time. Our work differs from Baca et al. in that we evaluate five SCA tools in terms of the source code metrics using the Juliet Test Suite.

Emanuelsson et al. [36] compared the performance of three SCA tools (Polyspace Verifier, Coverity Prevent and Klocwork Insight) using a set of applications from Ericsson. The authors in this study concluded that the Coverity and Klocwork tools have a high value of true positive warnings. In other words, both of these tools highlighted the most security vulnerabilities in the source code, while PolySpace tool did not. Importantly, both Coverity and Klocwork produced a low rate of false positive warnings, while PolySpace produced a high rate of false positive warnings. The work presented in this paper evaluates the SCA tools in term of software engineering metrics not only in terms of true positive and false positive warnings generated by these tools.

2.2 Classifying SCA Tools' Warnings

A number of studies have applied machine learning techniques on characteristics describing the warning generated by static code analysis tools.

Barstad et al. [37] investigated if they can predict the quality of the source code based on the static metrics' value (*e.g.*, McCabe Cyclomatic Complexity and Halsted metrics) using ML techniques (*e.g.*, Naive Bayes (NB), KNN, and decision tree). In their work, the source code was classified as "well written" or "badly written". Based on their results, the NB outperforms the other classifiers. Our work differs their work in three main ways. First, our work investigates the relationship between the SCA tools' warnings and the software engineering metrics. Second, we evaluate the proposed approach against seven CWEs using two SCA tools, while Barstad et al. used the SCA tools to compute the metrics value only. Lastly, they apply ML techniques to predict the source code quality; while in this work we predict how the SCA tool will behave on the given source code.

Yuksel et al. [38] proposed an approach to reduce the number of the false-positive warnings that are emitted by SCA tools by applying 34 ML techniques over datasets containing 10 different artifact characteristics. They conclude that the ML techniques can be a useful approach to classify the SCA tools' warnings because they achieved 87% accuracy. Our work is similar to their work in that we want to reduce the number of false-positive warnings. The main difference between their work and our work, however, is that we use the source code characteristic to predict the SCA tool behavior (*i.e.*, the SCA tool will generate true positive, false positive, or false negative warnings) on the source code.

Koc et al. [39] trained both a Bayesian classifier and a long short-term memories (LSTM) neural network on bytecode instructions to predicate the false positive warnings. In our work, we train our models on the source code, not on bytecode instructions, which are simplified and easier to analyze with ML techniques as compared to the source code. On the other hand, we evaluate which metrics are highly correlated with each type of warning generated by the SCA tool, while in their work the authors evaluated which source code structures force the SCA tools to generate false positive warnings.

Reynolds et al. [40] identified and documented 14 of different kinds of false positive patterns, by running three of SCA tools against C/C++ Juliet test suite. Then the authors reduced the source code manually in order to remove the unrelated instructions. In our work, we run a number of ML techniques and infer which of software engineering metrics are related to each type of SCA tools' warnings.

Lastly, Tripp et al. [41] tackled the problem of false-positive warnings by combining the SCA tool user interaction with ML techniques. For example, users classify some of

SCA tool warnings into either actionable or spurious. Based on the user input the ML techniques predict the remaining SCA tool warnings automatically. In our work, we do not consider user interaction to classify the SCA tool warnings. We will consider the user interactions in future work.

2.3 Ranking SCA Tools' Warnings

In this section, we discuss related work on SCA tool warning prioritization methods either the historical data or statistical analysis techniques.

2.3.1 History-Based Warning Prioritization (HWP)

In this subsection we describe some of the popular history-based warning Prioritization (HWP) techniques. These techniques rank the SCA tool warnings using the software change history for warnings removed during bug fixes [30] [42].

Kim and Ernst [30] proposed a history-based warning prioritization algorithm by mining warning fix experience recorded in the software change histories. The proposed algorithm was evaluated by running three of SCA tools (FindBugs, JLint, and PMD) on three programs (Columba, Lucene, and Scarab). The main idea of this study was if the warnings were eliminated by fix-changes, this indicated that the warnings were important because they reported a real bug in the software source code. The experimental results showed that the algorithm improves warning precision by 17%–67%.

Likewise, Ruthruff et al. [43] developed a logistic regression model to predict and rank whether a generated warning represents a real defect in the given source code. Screening methodology was used to build an effective model by removing the factors with low predictive power from the dataset. The authors selected 33 factors to generate the required models. These factors were extracted from FindBugs warnings descriptions, Google warning descriptions, file characteristics, source code warning histories, source code factors, and code churn factors.

Our technique differs from the above works in that we utilize characteristics derived from the source code associated with the collected warnings to rank the new warnings. Because the proposed framework uses the value of the source code metric to rank the warnings, the proposed framework might be more practical and better at classifying the SCA tool warnings. Furthermore, The main limitation of the above works is that they depend on information from software histories, such as warning fix activities. Also, the software developers' prior knowledge and practices can reduce the efficiency of these methods by increasing the likelihood of missing new potential defects that the developers have not encountered before. Likewise, these methods can be biased toward the SCA tool warnings that have a similar flavor [44].

To overcome this restriction and to build an unbiased model, we use Juliet test suite to build our model. Juliet test suite is a set of thousands of small test programs written in C/C++ to present over 100 classes of a common software weaknesses, such as deadlock and buffer overflow [45]

2.3.2 **Statistical Analysis-Based Ranking**

Kremenek and Engler [27] proposed z-ranking, a technique that ranks the warnings emitted by SCA tools by employing a simple statistical model. The main idea of this technique is that code containing many successful checks (safe cases analyzed by the tool) and a small number of warnings, tends to contain a real error.

On the other hand, Yungbum et al. [46] tackled the problem of false positive warnings by proposing an analyzer called Airac (Array Index Range Analyzer for C), which collects all the true buffer overrun points in ANSI C programs. This analyzer uses the Bayesian statistics to compute the probability that a warning indicates a true defect. Warnings are sorted by the probabilities value.

Likewise, Ribeiro et al. [47] ranks SCA tool warnings during software development, allowing developers to fix only the true positive warnings. To rank the warnings, the authors first ran three of open-source SCA tools (CppCheck, Clang Static Analyzer, and Frama-C) that supported C/C++ programming language on the source code to collect the warnings. They then trained a set of decision trees using AdaBoost to create a stronger classifier using a set of features obtained from the labeled aggregated warnings, such as tool name, number warnings in the same file, category(buffer, overflow, etc.), redundancy level, and number of neighbors. Finally, They used AdaBoost classifier probabilities to rank the warnings as either true positive or false positive. The experimental results showed that the generated classifier achieved 80% classification accuracy. This work differs from ours in two principal ways. First, our work ranks SCA tool warnings using a different of software engineering metrics (source code metrics); second, our datasets represent some of the most common software weaknesses, such as CWE-134.

The main limitation of these techniques is that they require a large number of labeled aggregated warnings to build and train the classifier. Producing these datasets requires a significant manual effort.

3. EVALUATION OF STATIC CODE ANALYSIS (SCA) TOOLS USING SOFTWARE ENGINEERING METRICS

In Chapter 1, we have presented why we need to evaluate the SCA tools and why we need to generate a number of datasets. In Section 2.1 of Chapter 2, we have already discussed the related research on evaluating SCA tools. In this chapter, we first describe the challenges associated with evaluating SCA tools in Section 3.1. Then, we formally present the Static Code Analysis Tool Evaluator (SCATE) framework, which is our novel contribution for evaluating SCA tools in term of software engineering metrics in Section 3.2. Section 3.3 illustrates the Case Study that we use to evaluate the proposed framework. In Section 3.4, we describe results of applying SCATE to the selected code base. Finally, we summarize our main contributions in Section 3.5.

3.1 Challenges Addressed by the Proposed Framework

We address several challenges in this chapter, including the following:

- A wide range of SCA tools is available to software developers and companies. These tools improve source code quality and increase software security by revealing potential security vulnerabilities early in the software development process [48]. Therefore, selecting the best tool or set of tools can pose a significant challenge because manual operation of the tool, as well as manual checking of a large number of generated reports, is necessary to determine which SCA tools are suitable for a given source code [15]. Manual SCA tool selection is an extremely time-consuming and tedious task [7].

- Most current research evaluates the effectiveness of SCA tools by confirming whether their use improves source code quality and software security. In other words, the tool that highlights the largest number of potential defects in the source code is considered the best. Few current studies, if any, investigate the effects of how the source code was written on the ability of the SCA tools' ability to identify potential defects [49] .

- Not all the SCA tools can identify the same source code weaknesses. Developers therefore find it difficult to decide which SCA tools are best for their source code by considering only the total number of uncovered security defects [48] .

To overcome these challenges, we extended the SCATE [1] framework, a framework for evaluating the quality of static code analysis tools.

3.2 **The Approach of SCATE**

This section discusses the design of the framework that we have created to evaluate the quality of SCA tools, which is illustrated in Figure 3.1. The framework is written in Python, and is designed to be extensible to any SCA tool we want to evaluate either locally or remotely. The framework is also designed to be extensible to different code bases used to evaluate an SCA tool. The key entities in the framework are as follows:

- **Command.** The Command is an interface that defines the different tasks and operations supported by the framework. Such commands currently include: parsing the source code; building the knowledge base from a source code base acting as the test suite; analyzing an SCA tool's report; and analyzing source code metrics like code complexity, dependency between the functions, and source lines of code (SLOC).

- **Tool.** The Tool is an interface that defines how SCA tool integrates with the framework. The Tool interface allows the framework to perform several key

[1]SCATE is an acronym for Static Code Analysis Tool Evaluator.

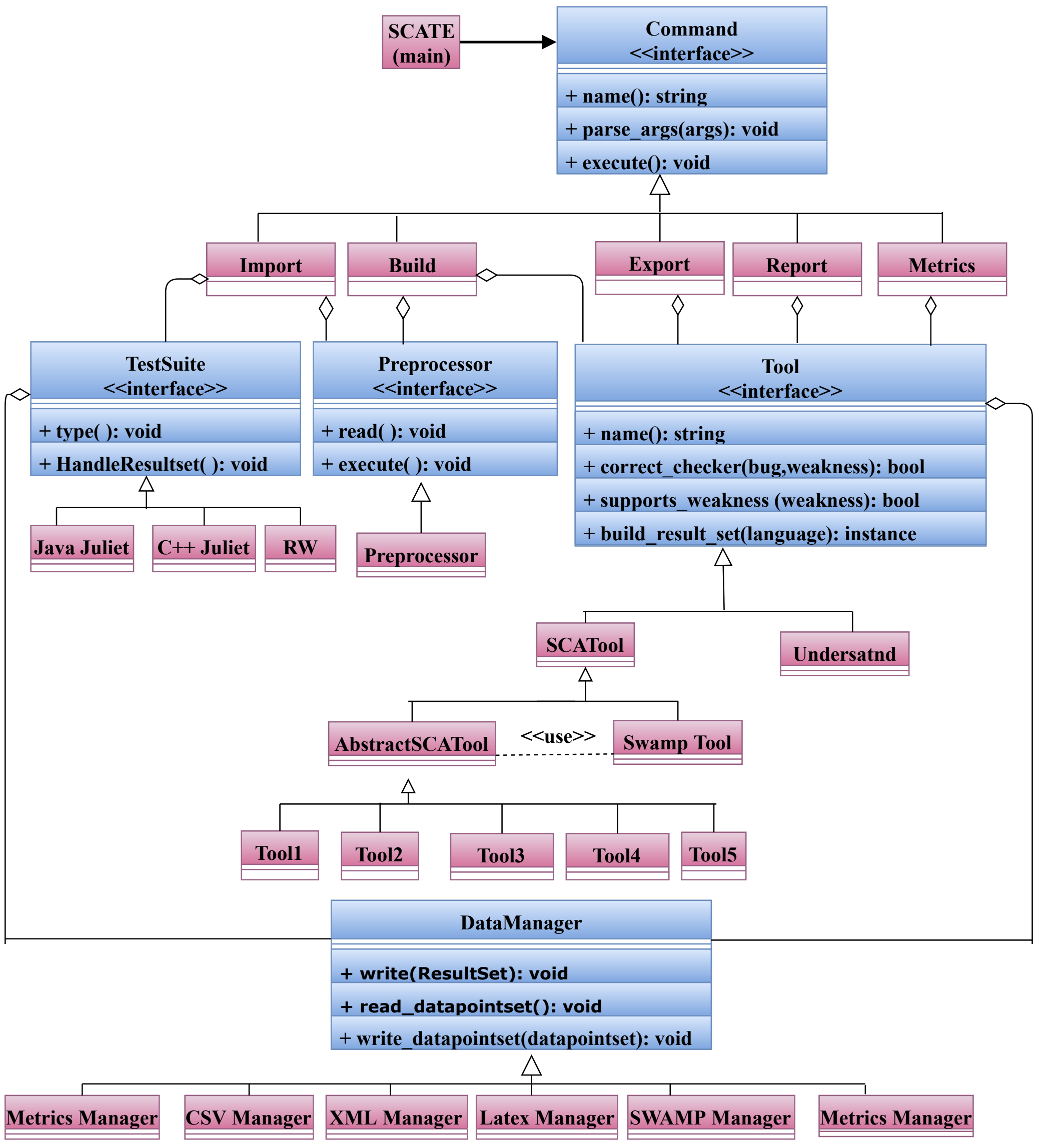


Fig. 3.1.: General design of our extensible framework
for evaluating static code analysis tools.

operations offered by SCA tool, such as checking if the SCA tool supports a specific weakness, and checking if the reported bug has the target type (*i.e.*, the type of flaw that the test case under testing targets).

- **DataManager.** The DataManager is used to define the output file format for each command in this framework. For example, the import command will use the DataManager to generate the knowledge base of the known flaws from the test cases. Likewise, the build command will use the DataManager to read the SCA tools' report and convert it into a hierarchical abstraction (see Figure 3.2).

Each SCA tool generates a different format of report which makes the comparison hard. To solve this problem we use the hierarchical abstraction to normalize the SCA tool's report.

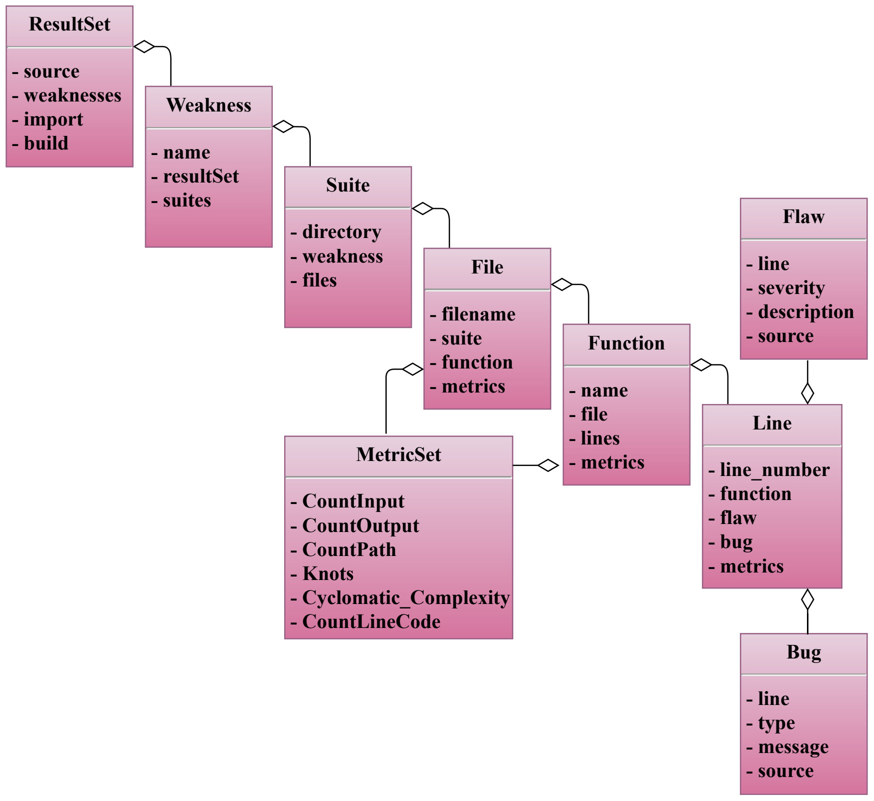


Fig. 3.2.: Entities that make the knowledge base for a Test Suite in the framework.

- **Preprocessor**. The Preprocessor is an interface that allows the framework to preprocess the source code and complete any information that will be missing from an SCA tool report, like the name of the function that contains the known flaws.

- **TestSuite.** The TestSuite is an interface for integrating different Test Suites into the framework. The Test Suite is then used by the framework to construct a knowledge base (or test oracle) from the code base. The knowledge base is then used to determine the true positive, false positive, and false negative rate of an SCA tool for the corresponding TestSuite.

3.2.1 **Commands Supported in the Framework**

Each command in the framework is responsible for a given task in evaluating the SCA tools. Currently, we have implemented the following set of commands:

- **Import.** The import command is used to create a knowledge base from the source code in a TestSuite. The source code must contain the tags identified in Table 3.1, which originate from the Juliet test suite, to generate the correct knowledge base that is comprised of the entities listed in Figure 3.2. However, we simulated these tags in real-world applications using the information available at (`https://www.cvedetails.com`).

Table 3.1.: The different Tags used by the Framework
to correctly label locations of interest.

Tag name	**Description**
POTENTIAL FLAW	Indicates a flaw that has the target type and it appears based on specific conditions.
INCIDENTAL FLAW	Indicates a flaw that may be detected, but is not the main focus of the test case.
FIXED	Indicates a place in the source code that originally had a flaw and is no longer present.

- **Build.** The build command executes different operations supported by a SCA tool. This includes executing the SCA tool against the Test Suite either locally or remotely; extracting results; and building an actionable knowledge base from the reported bugs.

- **Metric.** The metric command is used to compute the different software engineering metrics for each source file in a Test Suite. The software engineering metrics are then integrated back into the knowledge base. Currently, we are using the Understand (https://scitools.com/) tool to generate our software engineering metrics. Our framework, however, is not limited to only using Understand.

- **Export.** The export command computes the true positive, false positive, and false negative rate for the SCA tool by comparing its generated output against the constructed knowledge base. An output is identified as a true positive when the SCA tool correctly labels the flaw. The output is identified as a false positive in three situations: (1) when the SCA tool reports there is a flaw in the source code, but it really does not; (2) when the SCA tool reports the fix tag in the good function or in the good class as flaw; and (3) when the SCA tool reports the flaw tag in the bad function or in the bad class with wrong type. Lastly, an output is identified as a false negative when the SCA tool does report a known flaw in a bad function or bad class.

- **Report.** The report command converts the output from the export command into a human-readable report.

3.2.2 **Integrating With the SWAMP**

The SoftWare Assurance Marketrue positivelace (SWAMP) is a cloud environment for running source code against different static code analysis tools. The SWAMP provides 19 open-source SCA tool and 4 commercial SCA tools. Its SCA tools support five programming languages: C/C++, Java, Python, and Ruby. There are two ways to use the SWAMP. Either use it via their hosted cloud computing platform (mir-swamp.org), or use the SWAMP-in-a-Box(SiB) open-source distribution [50].

Integrating our framework with SWAMP allows the developer to evaluate a wide variety of SCA tools in the context of software engineering metrics. To perform this integration, we implemented the following two components:

- **SWAMPManager.** The SWAMPManager reads the SWAMP Common Assessment Result Format (SCARF) files, which is the common format the SWAMP uses to for reporting the results of an SCA tool, and builds the abstract model hierarchy (see Figure 3.2) for our framework.
- **SWAMPTool.** The SWAMPTool acts as a proxy for running SCA tools run remotely in the SWAMP.

3.2.3 Classifying SCA Tools Output

We faced several challenges when evaluating the quality of SCA tools. For example, many of the open-source SCA tools do not document the CWEs [51] their checkers identify, which is the single entity in an SCA tool responsible for identifying a single problem. This is important because it allows us to correctly identify when an SCA tool is generating a true positive, false positive, or a false negative. In our work, we use the following approach to classify an SCA tool error message as a true positive, false positive, or false negative:

- **True positive.** We consider an error message to be a true positive if the SCA tool highlights the predefined flaw of the target type in the correct location. For example, in the Juliet test suite the reported flaw should be either in the function or class with the word `bad` in its name. To better understand the true positive, Listing 3.1 shows a simplified code snippet from the Juliet test suite that causes Tool2 to generate a true positive. The code snippet uses the `data` pointer to print out the ASCII code of *"A"* in Hexadecimal using **printHexCharLine** function after delete the pointer using **delete**. Tool2 successfully detected this flaw [52].

```
//CWE416_Use_After_Free__new_delete_char_18.cpp
void bad(){
    char * data;
    // Initialize data
    data = NULL;
    goto source;
source:
    data = new char;
    *data = 'A';
    //POTENTIAL FLAW: Delete data in the source
    //the bad sink attempts to use data
    delete data;
    goto sink;
sink:
    //POTENTIAL FLAW: Use of data that may have
    //been deleted
    printHexCharLine(*data); // True Positive By Tool
    //POTENTIAL INCIDENTAL: Possible memory leak
    //here if data was not deleted
}
```

Listing 3.1: True positive example from CWE-416 test case

- **False positive.** We consider an error message to be an false positive if the SCA tool highlights a flaw with incorrect type or the fix tag as a flaw. Likewise, the SCA tool reports a flaw in the source code where in actual there is none. To better understand the false positives, Listing 3.2 shows a simplified code snippet from the Juliet test suite that causes Tool1 to generate a false positive. Tool1 generates an Uninitialized **dataPtr1** variable, but the source code assign the memory address of **data** in line 4 [53].

```
1  //CWE124_Buffer_Underwrite__char_declare_loop_32.c
2  void bad(){
3      char * data;
4      char * *dataPtr1 = &data;
5      char * *dataPtr2 = &data;
6      char dataBuffer[100];
7      memset(dataBuffer, 'A', 100-1);
8      dataBuffer[100-1] = '\o';
9      {
        //False Positive: Uninitialized Variable.
        char * data = *dataPtr1;
        //Flaw: Set data pointer to before the
        //allocated memory buffer
        data = dataBuffer - 8;
        *dataPtr1 = data;
    }
}
```

Listing 3.2: False positive example from CWE-124 test case.

- **False negative.** If the SCA tool does not identify the flaw for the corresponding weakness in a function or class that contains the word `bad` this flaw will be considered as an false negative. To better understand the false negatives, Listing 3.3 shows a simplified code snippet from the Juliet test suite that causes Tool3 to generate a false negative. Tool3 could not highlight the potential flaw in the source code. This tool did not examine the new value of the **data** before executing **printLine(100/data)** instruction. In other words, **bad** function calls **bad_source** function by passing the reference of **data** variable, then **bad_source** function uses a random function to assign new value to **data**. This new value may be **zero**, so when **bad** function use this value without checking if it equals **zero** a divide-by-zero weakness may be raise [54].

```
//CWE369_Divide_by_Zero__int_rand_divide_43.cpp
static void bad_source(int &data)
{
    data = RAND32();
}

void bad()
{
    int data;
    data = -1;
    bad_source(data);
    //POTENTIAL FLAW: Possibly divide by zero
    printIntLine(100 / data);
}
```

Listing 3.3: False negative example from CWE-369 test case.

3.3 Case Study

In this section the case study that we utilize in order to evaluate the effectiveness of the proposed framework.

3.3.1 Selected Code Base

In this study, we have used two different types of test suites: the synthesized (Juliet test suite) and real-world (Xerces-C++ and Apache Tomcat) source codes to evaluate the SCA tools.

In the first phase of the evaluation process, we ran each SCA tool against the Juliet Test Suite for either C++ or Java to evaluate their quality. The Juliet Test Suite was created by the National Security Agency's (NSA) Center for Assured Software (CAS) to evaluate any SCA tools' ability. The Juliet Test Suite for C++ programming language contains 61,387 test cases and covers 118 CWEs (Common Weakness Enumeration), aiming to create a catalog of software weaknesses and vulnerabilities [55]. On the other hand, the NIST Juliet suite for Java programming language contains 23,957 test cases and covers 113 CWEs [56]. In the second phase of the evaluation process, we ran the open-source SCA tools (Tool1, Tool4, and Tool5) against two of real-world applications, which are listed below:

1. **Xerces-C++** (http://xerces.apache.org/xerces-c/). An XML parser framework written in the C++ programming language; it can parse, generate, and validate XML documents using the DOM, SAX, and SAX2 APIs. It is one of the most widely used C++ XML parsers..

2. **Apache Tomcat** (tomcat.apache.org). An implementation of the Java Servlet [57] and JavaServer Pages [58] technology. One of the most widely used Java web-based application servers, Tomcat is embedded in many enterprise application servers that serve very high volumes of requests.

Table 3.2 shows important information about the real-world applications used as case studies in this study to evaluate open-source SCA tools in the context of software engineering metrics values.

Table 3.2.: Some important information about the real-world applications used as case study

Name	**Language**	**Number of vulnerabilities**	**Version**	**CWE**
Xerces-C++	C++	9	3.0.0	476, 20, 119
Apache Tomcat	Java	32	9.0.0.M1	22, 254, 284, 434,79

To explore the performance of open-source SCA tools on two of real-world applications, we determine potential security vulnerabilities and their locations using the information in the newer version, which has the fixed the vulnerabilities and the security reports available at the application website. On the other hand, to identify the type of the actual security vulnerabilities, we used the information available at (`https://www.cvedetails.com`) [59].

3.3.2 **Selected Static Code Analysis (SCA) Tools**

The proposed framework evaluates five SCA tools, which are listed below [2]:

- **Tool1.** An open-source SCA tool for C/C++ code. This tool uses Lexical Analysis to find the flaws in C++ source code. Lexical Analysis matches the tokenized source code with a list of checkers and reports if it finds a suspicious pattern. Ignoring the data flow of the source code causes the Tool1 to not able to detect vulnerabilities caused by the invalidated external input.

[2]At the time of writing this dissertation we keep the name of the SCA tools confidential because we do not have time to discuss the results with the SCA tools' vendors

- **Tool2.** A commercial SCA tool for C/C++ and Java code. This tool analyzes both the source code and binaries. To find flaws in C++ and Java source code, Tool2 builds an abstract model from the source code, then explores it with the symbolic execution engine to test every execution path.

- **Tool3.** A commercial SCA tool for C/C++, Java, and C# code. This tool analyzes the source code after building the control flow and data flow. Also, this tool provides a range of security checkers to detect the potential security vulnerabilities in the source code. However, Tool3 ranks the uncovered defects in order to consider only the four most severe levels of warnings to avoid false positive warnings problem.

- **Tool4.** An open-source SCA tool for Java code. This tool focuses on finding bugs or potential performance problems, not style or formatting errors, using a list of bug patterns and by using data flow analysis for source code. In contrast to the previous tools, this tool uses byte-code, not source code. The potential errors reported by Tool4 are classified into four ranks: "scariest" (rank 1-4), "scary" (rank 5- 9), "troubling" (rank 10-14), and "of concern" (rank 15-20). These ranks reflect the severity and impact of errors in the software.

- **Tool5.** An open-source SCA tool for Java and JavaScript code. Tool5 includes a set of built-in rules in order to detect common programming flaws such as unused variables, empty catch blocks, and so forth, as well as supporting the ability to write custom rules. Tool5 includes CPD (Copy-Paste Detector), which attempts to find the duplicated code in Java, C, C++, PHP, Ruby, FORTRAN, JavaScript, PLSQL, Python, and other programming languages.

3.3.3 **Selected Weaknesses (CWEs)**

Although the five SCA tools have been evaluated against 91 CWEs in the test cases. The results of the following CWEs have been discussed in detail:

- **CWE-369: Divide by Zero.** This weakness occurs when an unexpected value is provided to the product/calculation, or if an error occurs that is not properly detected [54]. To better understand this weakness, Listing 3.4 shows a simplified code snippet that contains a function that computes the average of student grades.

```
//CWE-369_Divide_by_Zero_example
float compte_avg(float total, float num_grads){
    // //POTENTIAL FLAW
    return total/num_grads;
}
```

Listing 3.4: CWE-369 Example

Without validating the parameter value (*num_grads*) used as the denominator is not zero a divide by zero error can be occurred. To avoid this error we just need to ensure that the input value of the *num_grads* will be always not zero.

- **CWE-457: Use of Uninitialized Variable.** This weakness occurs when the source code uses a variable that has not been initialized. This weakness may lead to unpredictable or unintended results [60]. To better understand this weakness, Listing 3.5 shows a simplified code snippet that contains a function that print out the value of the local variable.

```
//CWE-457_Use_of_Uninitalized_Variable_example
void print_out(){
    double dvalue;
    //POTENTIAL FLAW
    printDoubleLine(dvalue);
}
```

Listing 3.5: CWE-457 Example

The variable *dvalue* is never assigned any value before using it. The SCA tool should be able to identify this error as CWE-457 warning message.

- **CWE-476: NULL Pointer Dereference.** This weakness occurs when the application dereferences a pointer that it expects to be valid, but it is NULL. This weakness may lead to crash or exit.

```
//CWE-476_NULL_Pointer_Dereference_example
void bad(){
        int *ptr = NULL;
        /* Potential FLAW: Dereferencing of the null pointer 'ptr' */
        if (*ptr == 17)
        cout << ("ptr = 17") << endl;
}
```

Listing 3.6: CWE-476 Example

Listing 3.6 shows that the analyzer has to identify the fragment of code that uses a null pointer. In the if condition, there is a logical error that leads to dereferencing of the null pointer. The error may be introduced into the code during code refactoring or through a misprint [61].

- **CWE-382: J2EE Bad Practices: Use of System.exit().** This weakness occurs when access to a function that can shut down the application by calling System.exit().

- **CWE-484: Omitted Break Statement in Switch.** This weakness occurs when the source code omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. However, when the software developers intended to execute code associated with one condition this weakness can cause some problems.

3.3.4 Selected Software Engineering Metrics

The software engineering metrics we used to evaluate the SCA tools are a set of source code metrics, including Volume, Object-Oriented, and Complexity metrics. To compute the value of the software engineering metrics from the source code, we ran the Understand tool, which is a SCA tool designed to compute the values of most traditional software engineering metrics for C++ and Java programs. The Understand tool supports more than 39 software engineering metrics. Moreover, a different number of techniques can be used to acquire metrics from the source code, either by using the source code tokens, a data-flow graph, or a control-flow graph. The metrics that were employed in this work are categorized as below [62]:

1. **Basic Count Line Metrics.** These metrics retrieve information about each line in the source code in the scope of a function, a class, or a file. Within this family of metrics, we used CountLineBlank, CountLineCode, CountLineComment, CountLineCodeDecl, CountLineCodeExe, CountLineInactive, and CountLinePreprocessor.

2. **Basic Count Metrics.** These metrics are divided into two main sets: CountDeclClass, which retrieve the number of classes in the file; and, CountDeclFunction, which retrieve the number of functions declared in the file.

3. **Basic Token Metrics.** These metrics retrieve information about the source code complexity. Within this family of metrics, we used Cyclomatic, CyclomaticModified, CyclomaticStrict, Countsemicolon, and MaxNesting.

4. **Control Flow Metrics.** These metrics are computed from the control flow graph of the function. Within this family of metrics, we used Knots, Essential, MinEssentialKnots, MaxEssentialKnots, and CountPaths.

5. **Miscellaneous.** We used CountInput and CountOutput metric. These metrics are generally estimated at three levels: project-level, file-level, and function-level.

The software engineering metrics are generally estimated at four levels: project, class, file, and function. In this work, we estimated the software engineering metrics at the class, file, and function levels only [62]. However, we will discuss the following metrics in detail [62]:

1. **Essential Complexity.** This metric computes the source code complexity after iteratively replacing all the well-structured control structures, such as the if-then-else and while loops, with a single statement. The Essential complexity is given with the following equation.

$$\begin{aligned} Essential = (&number\_of\_jumps\_for\_each\_node\_ \\ &that\_has\_multiple\_children - 1) + 1 \end{aligned} \tag{3.1}$$

2. **Cyclomatic Complexity.** This metric computes the source code complexity by using the McCabe Cyclomatic technique [63], where the complexity of any structured source code with only one entrance and one exit point is equal to the number of decision points contained in that source code, plus one.

3. **CountPathLog.** This metric belongs to the category of Complexity Metrics and computes the logarithm of the total number of unique paths in the function by excluding the abnormal exits and the **GoTo** statements. This software engineering metric is assessed by the Understand static tool at the function level.

4. **CountOutput.** This metric belongs to the category of Object-Oriented metrics and computes the number of outputs of the function in source code. The outputs may be classified into functions calls, parameters set/modify, and global variables set/modify. We computed the value of this software engineering metric at function or method level by running the Understand static tool, which follows the information approach of the Fan-Out to calculate the value of the CountOutput metric.

5. **CountPath.** This metric belongs to the category of Complexity Metrics and computes the total number of unique and possible paths in the function by excluding the abnormal exits and the **GoTo** statements. This software engineering metric is assessed by the Understand static tool at the function level.

6. **Knots.** This metric belongs to the category of Complexity metrics, and is a measure of overlapping jumps. In the given source code, the Knots value equals the number of line-crossings that determine where every jump in the flow of control occurs.

7. **MaxNesting.** This metric belongs to the category of Complexity Metrics, and computes the complexity of the given function or method in terms of the level of the control constructs, such as if, while, for, and switch in the function or method.

8. **CountInput.** This metric belongs to the category of Object-Oriented metrics and computes the number of inputs the function uses. The inputs may be classified into the function calleby, global variables used in the function, and the in parameters used in the function. We compute the value of this software engineering metric at the function or method level by running the Understand static tool, which follows the information approach of the Fan-In to calculate the CountInput metric value [62].

9. **CyclomaticStrict.** This metric computes the source code complexity, which equals the Cyclomatic Complexity metric with logical ANDs and ORs in the conditional expressions, and also adds 1 to the complexity for each occurrences.

10. **RatioCommentToCode.** This metric computes the ratio of the number of comment lines to the number of code lines in a given source code. In some cases, the value of this metrics is higher than 100, because some lines in the source code contains both the code and comments.

11. **MinEssentialKnots (MinKnots).** This metric reflects the minimum value of Knots metric that computed after all the structured programming constructs have been removed from a given source code.

12. **CyclomaticModified.** This metric computes the source code complexity, which equals the Cyclomatic Complexity metric except that each decision in a

multi-decision structure (such as switch) statement is not counted and instead the entire multi-way decision structure counts as 1.

13. **CountStmtExe.** This metric counts the number of executable statements in a given source code. A line in the source code can be executable and declarative, but a statement must be one or the other (or empty).

14. **CountStmtDecl.** This metric counts the number of declarative statements in a given source code.

15. **CountStmt.** This metric counts the number of declarative plus executable statements, given with the following equation:

$$CountStmt = CountStmtDecl + CountStmtExe + CountStmtEmpty \tag{3.2}$$

16. **CountSemicolon.** This metric counts the number of semicolons in a given source code.

17. **CountLineCodeExe.** This metric is used to compute the number of lines that contain executable source code.

18. **CountLineInactive.** This metric is used to compute the number of lines that are inactive from the view of the preprocessor.

19. **Preprocessor Lines.** This metric is used to compute the number of preprocessor lines.

20. **MaxEssentialKnots (MaxKnots).** This metric computes the max value of the Knots metric after the structure programming constructs have been removed in the given source code.

21. **CountLineComment (aka CLOC).** This metric reflects the number of lines containing a comment. This metric can overlap with other code-counting metrics.

22. **CountLine (aka NL).** This metric counts the number of physical lines in a given source code.

23. **CountLineCode (aka LOC, SLOC).** This metric reflects the total number of lines that contain source code, but only in a given function. For classes, the value of this metric will be the sum of the CountLineCode for the member functions.

24. **CountLineCodeDecl.** This metric counts the number of lines containing declarative source code. A line can be declarative and executable at the same time.

25. **CountLineBlank (aka BLOC).** This metric counts the number of blank lines in a given source code, excluding in inactive regions.

26. **CountDeclFunction.** This metric reflects the number of functions in the file.

27. **AltCountLineBlank.** This metric reflects the number of blank lines, including in inactive regions.

28. **AltCountLineComment.** This metric counts the number of lines containing comments, including comments within inactive regions.

29. **AltCountLineCode.** This metric counts the number of lines containing source code, including inactive regions.

30. **CountDeclClass.** This metric counts the number of classes in the file.

31. **CountDeclClassMethod.** This metric counts the number of static class methods.

32. **CountDeclClassVariable (aka NV).** This metric counts the number of class variables.

33. **CountDeclInstanceMethod (aka NIM).** This metric counts the number of instance methods.

34. **CountDeclInstanceVariable (aka NIV).** This metric counts the number of instance variables.

35. **CountDeclMethod.** This metric counts the number of local (not inherited) class methods.

36. **CountDeclMethodDefault.** This metric counts the number of local default visibility methods.

37. **CountDeclMethodPrivate (aka NPM).** This metric counts the number of local (not inherited) private methods.

38. **CountDeclMethodProtected.** This metric counts the number of local protected methods.

39. **CountDeclMethodPublic (aka NPM).** This metric counts the number of public methods, but only local (not inherited) methods.

3.4 Experimental Evaluation of SCATE

This section discusses our experimental results for evaluating SCA tools in the context of different software engineering metrics.

3.4.1 Experimental Setup

We used the Juliet test suites to perform our experiments against the SCA tools. To setup and execute our experiment, we executed the following steps:

- We used the import command to parse the source files in the test suite (Juliet test suites and real-world applications) and build a knowledge base with the ground truth. The ground truth contains information about the flaws, such as function name and line number, labeled in the test cases. In this step, the framework parses approximately 61,000 C++ or Java files depending on what SCA tool we are targeting for evaluation.

- We ran the SCA tools either locally or remotely in the SWAMP against the source code in the Juliet test suite and real-world applications. We then capture the generated output of the SCA tool because we need this to evaluate if the SCA tool is correctly labeling the flaws in the source code.
- Next, we assessed the value of the software engineering metrics for each CWE by executing Understand against the source code for the corresponding test cases.
- Last, we compared the ground truth with the output generated by the SCA tool, and assessed the performance of the SCA tool with respect to the different software engineering metrics.

We applied the steps above against the following SCA tools: an open-source SCA tool that supports C++ programming language (aka Tool1), two open-source SCA tools that support Java programming language(aka Tool4 and Tool5), and two commercial SCA tools that support C++ programming language(aka Tool2 and Tool3). The proposed framework, however, is not limited to only using these tools.

3.4.2 **Experimental Results for SCATE**

Although the five SCA tools have been evaluated against 91 CWEs in the test cases. The results of the following CWEs have been discussed in detail:

- **CWE-369: Divide by Zero.**
- **CWE-382: J2EE Bad Practices: Use of System.exit().**
- **CWE-457: Use of Uninitialized Variable.**
- **CWE-484: Omitted Break Statement in Switch.**
- **CWE-476: NULL Pointer Dereference.**

Following the behavior of SCA tools based on a set of software engineering metrics have been discussed in detail.

3.4.2.1 **CountOutput (Fan-Out)**

Figure 3.3 shows the results of the SCA tools for CWE-369 based on the values of the CountOutput (Fan-Out) metric. As shown in this figure, Tool3 initially finds more flaws than the other tools when the value of the CountOutput metric is low.

On the other hand, as the value of the CountOutput metric increases, which means the functions in the source code have high degree of coupling, Tool3 finds fewer flaws than Tool2 and the other tools. Likewise, Tool2 finds more flaws than the other tools when the value of the metric increases. Unfortunately, none of the tools can find any flaws in the source code when the value of the metric becomes more than six.

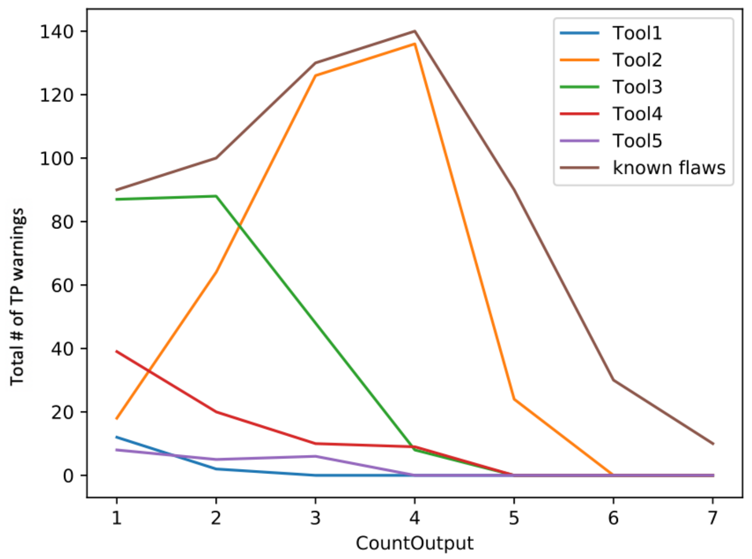


Fig. 3.3.: The behavior of SCA tools based on CountOutput for CWE-369.

However, based on the results shown in Figure 3.3 we can conclude that the SCA tools cannot understand the source code of function f when there are a large number of functions that depend on it or when function *f* heavily uses global variables. Therefore, the tools can miss some of the potential defects in the source code. To improve the ability of the SCA tools the software developers could rewrite their source code to reduce the coupling degree. For example, they can avoid using the global variables in their source code.

3.4.2.2 **CountInput (Fan-In)**

Figure 3.4 illustrates the behavior of the SCA tools for CWE-457 based on the CountInput. As shown in the figure, the number of uncovered flaws by both Tool2 and Tool3 decrease as the value of the CountInput increases. Tool2 finds more flaws than Tool3 when the source code has a high degree of Fan-In. In other words, when the functions in the given source code have a high number of calling functions and global variables read, Tool2 performs better than Tool3.

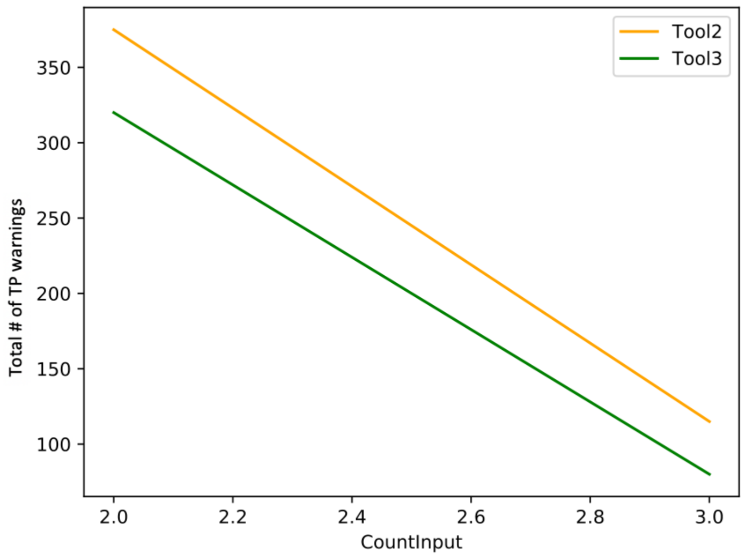


Fig. 3.4.: The behavior of SCA tools based on CountInput for CWE-457.

However, only Tool2 and Tool3 support this weakness so these tools will generate a number of warnings which can be either true positives or false positives. Therefore, we can use these warnings to analyze the behavior of these tools only.

3.4.2.3 **Lack of Cohesion in Methods (LCOM/LOCM)**

The Understand static tool uses the Chidamber and Kemerer method to compute the value of the LCOM metric [62]. For our experiments, Understand computed the value of this software engineering metric at the class level. Figure 3.5 illustrates the behavior of the SCA tools for CWE-382 based on the LCOM.

Only Tool4 and Tool5 support this weakness so these tools will generate a number of warnings which can be either true positives or false positives. Thus, we can use these warnings to analyze the behavior of these tools.

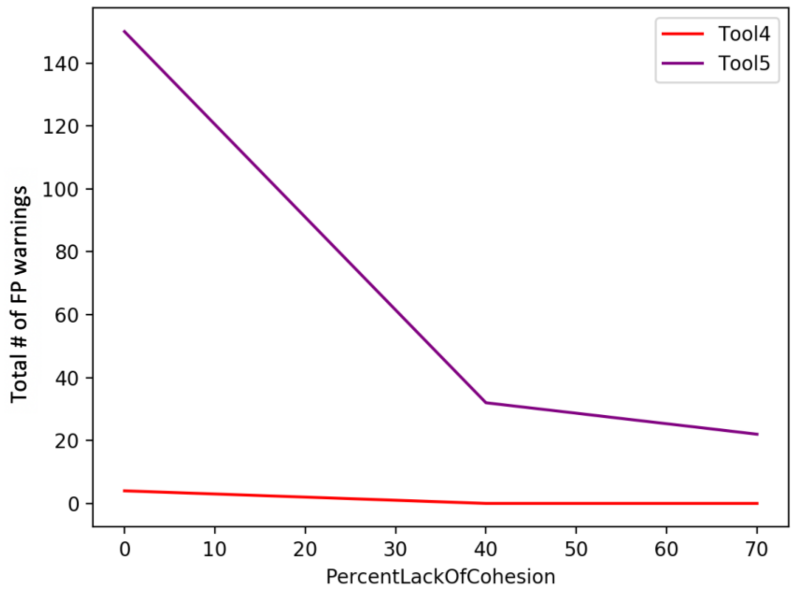


Fig. 3.5.: SCA tools behavior based on LCOM for CWE-382.

As shown in this figure, the number of fake warnings generated by both Tool4 and Tool5 decrease as the value of the LCOM metric increases. Tool5 generates more false positive warnings than Tool4 when the class has a high degree of LCOM. In other words, when the class in the given source code has a low degree of cohesion, Tool4 performs better than Tool5.

Likewise, Figure 3.6 showcases the behavior of the SCA tools for open-source software project based on the LCOM. From this figure, we can observe that Tool5 does not generate any fake warnings for the given source code. On the other hand, the number of generated fake warnings by Tool4 increases as the value of the LCOM metric increases. From these figures, we cannot conclude that Tool4 is better than Tool5. Likewise, we cannot conclude that Tool5 is better than Tool4. It depends on the structure of the given source code.

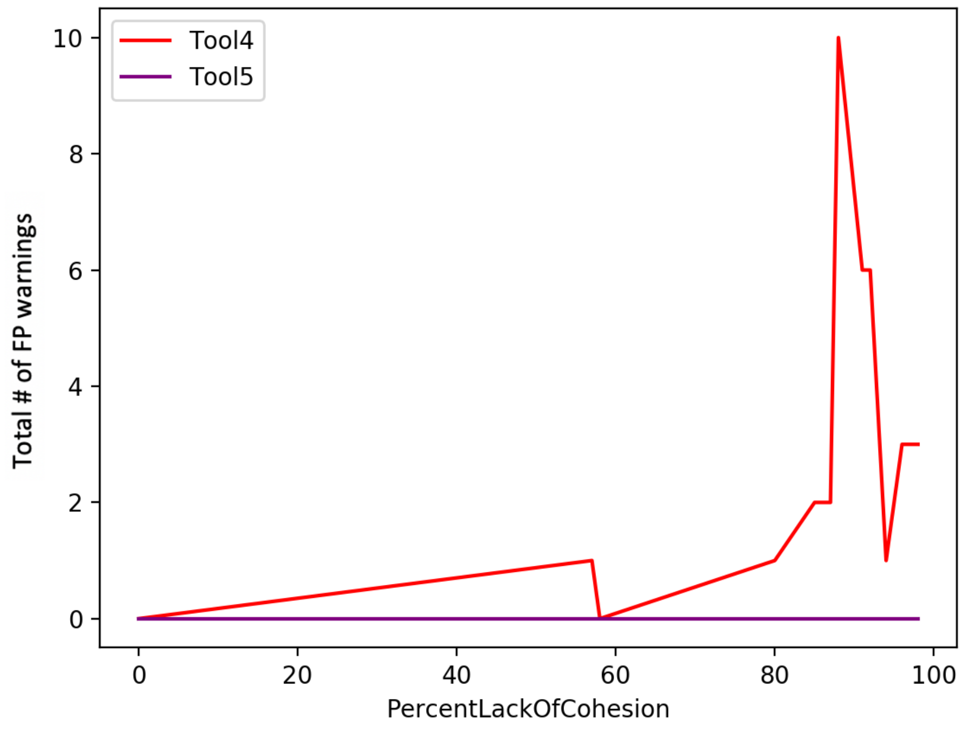


Fig. 3.6.: SCA tools behavior based on LCOM for Open-Source software

3.4.2.4 **MinEssentialKnots (MinKnots)**

Figure 3.7 illustrates the behavior of the SCA tools for CWE-484 based on the MinEssentialKnots. As shown in this figure, Tool5 generates a large number of false positive warnings when the given source code has a low number of overlapping jumps after all the structured programming constructs have been removed. On the other hand, Tool4 does not emit any false warnings for this weakness. Furthermore, the number of false positive warnings generated by Tool5 declines as the value of the MinEssentialKnots metric increases.

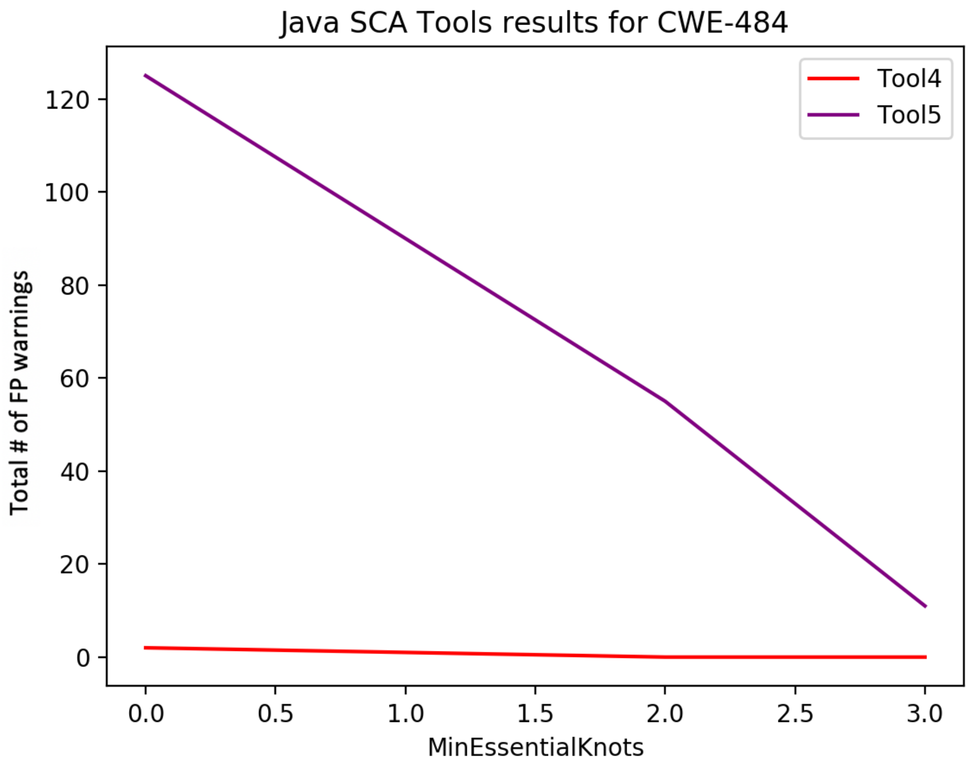


Fig. 3.7.: SCA tools behavior based on MinEssentialKnots for CWE-484

On the other hand, Figure 3.8 showcases the behavior of the SCA tools for Java real-world application based on the MinEssentialKnots. From this figure, we can observe that SCA tools do not behave in the same way when we use the real-world source code. This can be observed from the number of the false positive warnings generated by both the SCA tools. In other words, Tool4 does not generate any false positive warnings for the synthetic source code, while for the real-world source code it reports a number of false positive warnings when the value of the MinEssentialKnots metric was less than 13. However, the number of false positive warnings generated by Tool4 and Tool5 for the real-world application decreases as the value of the MinEssentialKnots increases. However, the developers can reduce the amount of knots in their source code by removing the misuse of "*break*", "*continue*", "*goto*", or "*return*" to improve the SCA tool performance.

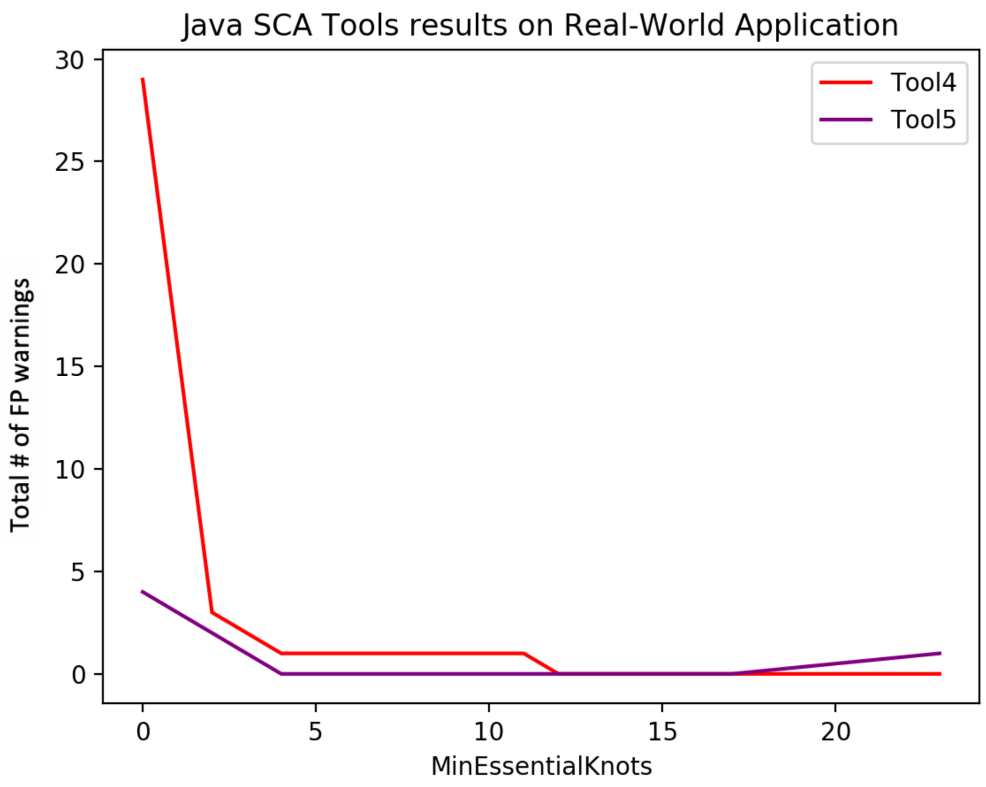


Fig. 3.8.: SCA tools behavior based on MinEssentialKnots for Open-Source software

3.4.2.5 **CountPath**

Figure 3.9 showcases the behavior of Tool1[3] for CWE-476 based on the value of the CountPath metric. As shown in this figure, Tool1 finds more flaws when the source code either does not include any control constructs and decision structures or includes a low number of unique paths. As the number of paths in the source code increases, Tool1 cannot find more of the potential flaws.

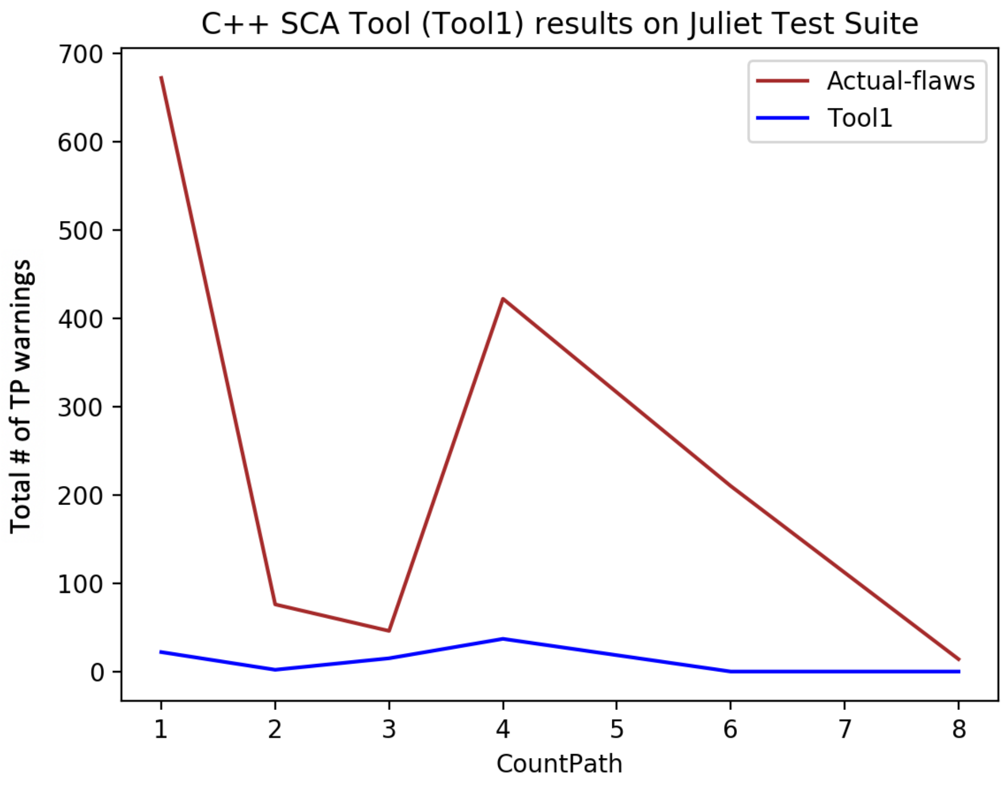


Fig. 3.9.: SCA tools behavior based on CountPath for CWE-476

Contrary to Figure 3.9, Figure 3.10 displays that Tool1 finds most of the potential defects in the real-world source code when the value of the CountPath metric increases.

[3] Since, currently, we do not have the license for the commercial SCA tools we decided to use the only Tool1, which is an open-source tool to compare the behavior of SCA tools using Juliet test suite and open-source software

This can be informed from the number of the true positive warnings for Tool1 when the value of the CountPath metric is higher than 100. On the other hand, Tool1 misses most of the defects when the source code has a low number of unique paths.

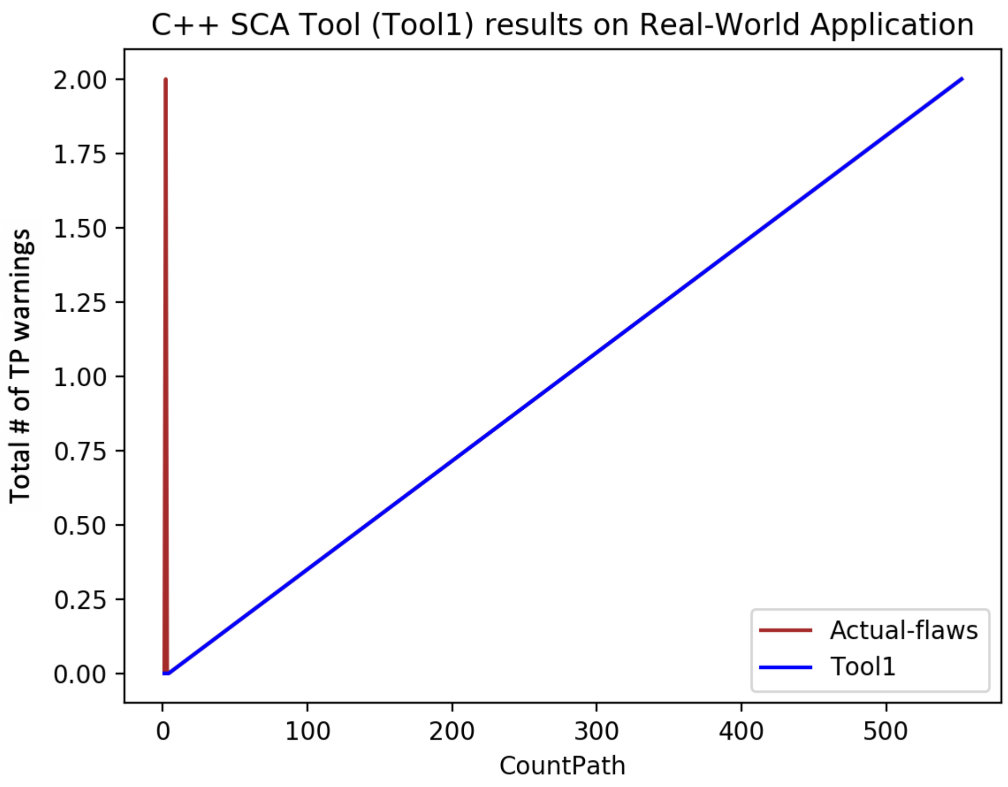


Fig. 3.10.: SCA tools behavior based on CountPath for Open-Source software

From the above-mentioned results, we can conclude that selecting the best SCA tool for our source code depends on the structure of the source code and the type of weakness that we want to uncover. Furthermore, the source code of the Juliet test suite is relatively simple compared to the source code of the real-world applications in terms of the number and types of the loops, control structures, and function calls that used in the real-world applications. This may force the SCA tool that generates a large number of either true or false positive warnings for the source code in the Juliet test suite to not report any fake or true positive warnings for the source code of the real-world applications.

3.4.3 **Threats to Validity**

For this framework, the main threat to external validity is most SCA tools do not have an easily accessible mapping of CWEs to checkers, which we use to classify the tools' reported bugs as true positive, false positive, or false negative. This threat causes many of the reported bugs considered as an false negative not as true positive and in sometimes increases the number of false positive.

Another threat to validity is that the evaluation performed in this chapter should, however, be replicated using many other open-source software projects in order to draw more general conclusions. We investigated the relationship between software engineering metrics and SCA tool behavior using only two open-source software projects, which is a relatively small number of projects.

3.5 **Summary of Contributions**

In this chapter, we have presented the Static Code Analysis Tool Evaluator (SCATE), which is a framework to evaluate SCA tools using synthetic source code in term of software engineering metrics. The following are the key contributions of the SCATE.

- Providing an extensible framework for evaluating SCA tools;
- Evaluating two commercial SCA tools and three open-source SCA tools in terms of software engineering metrics (source code metrics) against the Juliet test suite [55] [56] and real-world applications; and
- Discussing how software engineering metrics (such as Coupling and Cyclomatic Complexity) impact the true positive, false positive, and false negative rates of the SCA tools.

4. CLASSIFICATION OF STATIC CODE ANALYSIS (SCA) TOOL WARNINGS

In Chapter 1, we have presented why we need to classify the SCA tool warnings. In Section 2.2 of Chapter 2, we have already discussed the related research on classification SCA tool warnings. In this chapter, we first describe the challenges associated with classifying SCA tools in Section 4.1. Then, we formally present the Static Code Analysis Tool Warnings Classification (SCATWC) framework, which is our contribution for classifying SCA tool warnings using software engineering metrics in Section 4.2. Section 4.3 illustrates the Case Study that we use to evaluate the proposed framework. In Section 4.4, we describe results of applying SCATWC to Juliet test suite. Finally, we summarize our main contributions in Section 4.5.

4.1 Challenges Addressed by the Proposed Framework

Developing ML prediction model to classify SCA tool warnings and predict future warnings is challenging. This section presents some of these challenges that we tried to address. In short, we encountered the following challenges:

- ML techniques have been widely used to build prediction models to classify and predict SCA tool warnings. To train an ML classifiers, many current studies have derived a number of features from the meta-data of SCA tools' warnings, such as the file name, function name, line number, and total number of warnings generated for a given function. However, to improve the performance of the classifier we want to build, we have to use features that actually reflect the syntax of the source code that forces the SCA tool to generate a true positive, false positive, or false negative warning. Therefore, using the meta-data of the generated warnings to train a classifier will diminish its accuracy since those features do not truly reflect the source code's syntax.

- The availability of datasets has always been a constraint in research predicting SCA tool warnings and reducing false positive warnings. In other words, a number of professionals are usually hired to manually annotate the warnings and source code as true positive, false positive, or false negative. Annotating datasets in this way is considered a time-consuming process and is prone to inaccurate labeling.

- The main issue in predicting and classifying SCA tool warnings is that the underlying training dataset suffers from an imbalanced distribution problem, in that the training set for false positive and false negative warnings (the majority classes) is far larger than the training set of the true positive warnings (the minority class). This issue leads the classifier to correctly classify and predict all of the SCA tool warnings from the majority classes (false positives and false negatives) but to misclassify most of the SCA tool warnings of the minority class (true positives) [64] [65].

To overcome these challenges, we proposed the SCATWC [1] framework, a framework for classifying the warnings of SCA tools.

4.2 **The Approach of SCATWC**

In order to classify the warnings created by SCA tools based on the software engineering metrics (*i.e.*, source code metrics) new datasets have been created and a set of ML techniques have been utilized in the proposed approach. Fig. 4.1 indicates the overview of our approach. As shown in this figure, our approach includes two stages.

[1]SCATWC is an acronym for Static Code Analysis Tool Warnings Classification.

In the first stage, we generate a number of datasets by analyzing a given source code in two different ways; first, we compute the software engineering metrics such as Volume, Complexity, and Object-Oriented Metrics metrics. Second, we extract the SCA tools' warnings by utilizing a framework SCATE [7].

In the second stage, we utilize four of the common ML techniques. Our proposed approach, however, is not limited to only the four ML techniques discussed in this chapter.

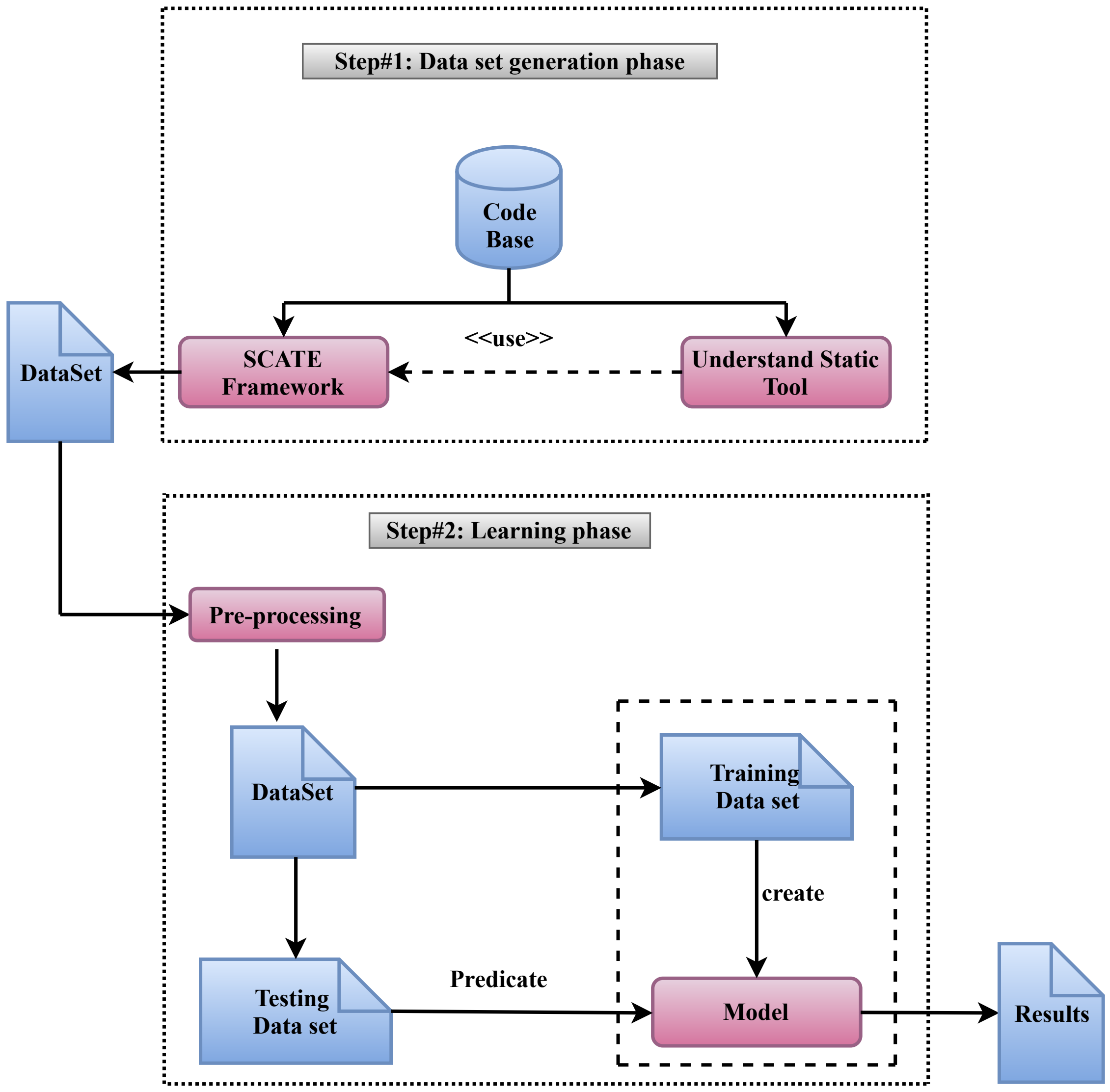


Fig. 4.1.: The overview of the proposed approach.

4.2.1 **Dataset Generation Stage**

In order to apply the ML techniques on a given source code, we have to follow a set of important steps:

1. **Extract SCA tools' warnings.** To extract the class values (*i.e.*, true positive, false positive, and false negative), we use SCATE [7], which is an open-source framework for evaluating the quality of a SCA tool based on the number of true positives, false positives, and false negatives it generates. This framework was extended to evaluate more open-source and commercial SCA tools either by running them locally or remotely in the SWAMP [50] (see Chapter 3).

   The main reason for using this framework in our research, is to run the SCA tools against Juliet test suite for the C++ language to highlight source code snippet that causes the SCA tool to:

   (a) Uncover the target security flaws in either the bad function or in the bad class implementation (aka true positive tag),

   (b) Report that there is a flaw while in reality there is not one (aka false positive tag), or

   (c) Report that there is no flaw while in reality there is one (aka false negative tag).

   These tags will represent class variable value in the generated dataset. Furthermore, the highlighted source code snippets will be used as inputs to the next step.

2. **Compute software engineering metrics.** To extract the most important attributes, or features, from a given source code, we integrated SCATE with the Understand tool (`https://scitools.com/`). Understand is a static analysis tool focused on source code comprehension, metrics, and standards testing.

   We use the Understand tool by executing it against the highlighted source code snippet from the Juliet test suite to compute different software engineering metrics at the function-level. The software engineering metrics (see Section 4.3.6) are then integrated back into the SCA tool report managed by SCATE. Our proposed approach, however, is not limited to only using the Understand tool, but is generic in nature.

Figure 4.2 showcases how we extended the design of the SCATE framework to achieve the dataset generation stage by adding a new command named **DSGenerating command**. The DSGenerating command is used to generate a number of datasets by comparing and integrated the outputs of the above-mentioned steps.

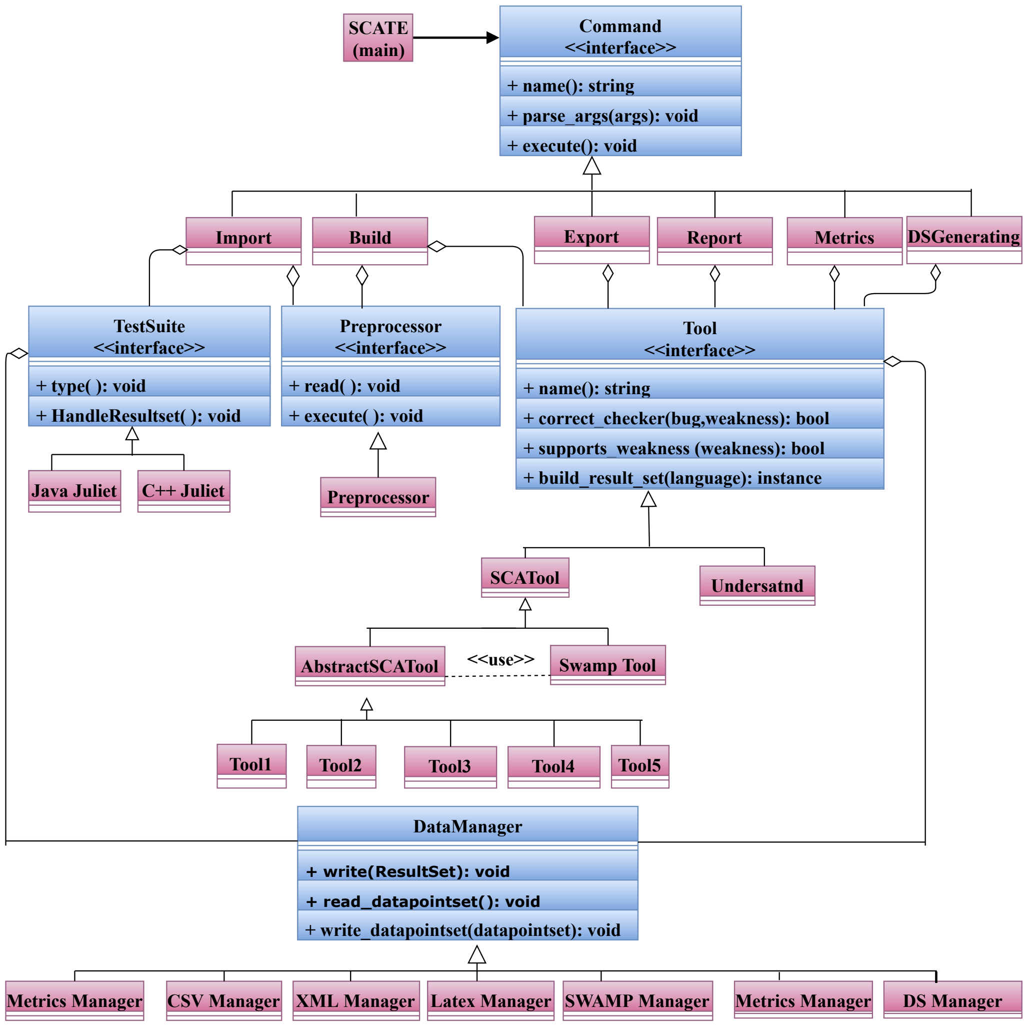


Fig. 4.2.: The new design of SCATE framework

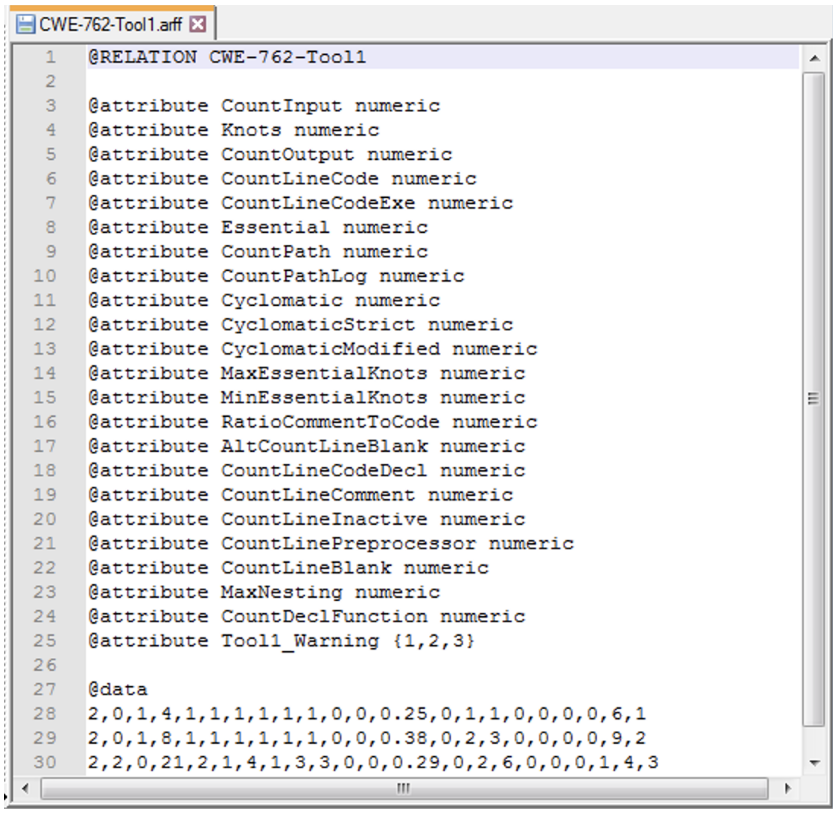
The output of this stage is a numerical dataset, which corresponds to the contents of a data matrix, where every column of the matrix represents a particular software engineering metric. The last column, however, represents a class value (*i.e*, true positive (which is represented by 1), false positive (which is represented by 2), and false negative (which is represented by 3)). Likewise, each row in the matrix corresponds to a given function in the source code. The dataset is, therefore, a multi-class dataset. However, in some special cases, when the SCA tool generates only two type of tags, or classes, the generated dataset will be a binary dataset.

To better understand the structure of the generated dataset, Figure 4.3 shows a sample dataset that is generated as an Attribute-Relation File Format (ARFF) file. From Figure 4.3, we can observe that the ARFF file contains two main sections: the header and the data section. The header section contains the name of the relation (in our example **CWE-762-Tool1**); a list of the attributes (software engineering metrics and whether Tool1 will generate true positive (1), false positive (2), or false negative (3) warning), and their types. In the data section, each line represents a function in a given source code, and each line contains both the values of the software engineering metrics for that function and what the warning Tool1 generated.

4.2.2 **Learning Stage**

The main goal of this stage is to build a classifier (*i.e.*, statistical model) by applying a set of ML techniques to the datasets generated in Stage 1 (see Section 4.2.1). The classifier is then used to predict a classification value for unknown source files. Another important goal of this stage is to learn which software engineering metrics are correlated with true positive, false positive, and false negative warnings generated by a SCA tool. This stage includes three phases:

1. **Preprocessing.** The generated dataset contains a set of data points representing a function that will cause the SCA tool to generate a true positive, false positive, or false negative message. The data points are represented by a collection of



```
@RELATION CWE-762-Tool1

@attribute CountInput numeric
@attribute Knots numeric
@attribute CountOutput numeric
@attribute CountLineCode numeric
@attribute CountLineCodeExe numeric
@attribute Essential numeric
@attribute CountPath numeric
@attribute CountPathLog numeric
@attribute Cyclomatic numeric
@attribute CyclomaticStrict numeric
@attribute CyclomaticModified numeric
@attribute MaxEssentialKnots numeric
@attribute MinEssentialKnots numeric
@attribute RatioCommentToCode numeric
@attribute AltCountLineBlank numeric
@attribute CountLineCodeDecl numeric
@attribute CountLineComment numeric
@attribute CountLineInactive numeric
@attribute CountLinePreprocessor numeric
@attribute CountLineBlank numeric
@attribute MaxNesting numeric
@attribute CountDeclFunction numeric
@attribute Tool1_Warning {1,2,3}

@data
2,0,1,4,1,1,1,1,1,1,0,0,0.25,0,1,1,0,0,0,0,6,1
2,0,1,8,1,1,1,1,1,1,0,0,0.38,0,2,3,0,0,0,0,9,2
2,2,0,21,2,1,4,1,3,3,0,0,0.29,0,2,6,0,0,0,1,4,3
```

Fig. 4.3.: Dataset sample of Tool1 warnings for CWE-762

software engineering metrics that measure a function's complexity, coupling, function's cohesion, and other metrics. Unfortunately, a SCA tool can generate more than one warnings (*i.e.*, false positive, false negative, and true positive) for the same function. This can result in contradicting data points in the generated datasets. To address this problem, we remove the contradicting data points as Hernández et al. [66] suggest.

2. **Model Learning.** This phase involves constructing a classifier by using the training dataset and by examining four of the ML techniques: Support Vector Machine (SVM) [67], K-Nearest Neighbor (KNN) [68], RF [69], and Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [70].

3. **Prediction.** The generated classifiers (from the model Learning phase) are used to predict whether a function in unknown source code will cause a SCA

tool to generate a true positive, false positive, or false negative message based on the value of software engineering metrics for the corresponding function.

4.3 **Case Study**

We use the following case study to evaluate the effectiveness of the proposed approach.

4.3.1 **Selected Machine Learning (ML) Techniques**

In this section, we briefly discuss the ML techniques used in this chapter.

4.3.1.1 **Feature Selection**

In this work, feature selection is important because it filters redundant and the inefficient software engineering metrics. We use the Correlation-based Feature Selection (CFS) technique [71] [72] to identify the most significant software engineering metrics. The CFS technique searches all the combinations of the software engineering metrics to find the best combination of the metrics [71].

The CFS technique evaluates the correlation between the software engineering metrics and the class. The selected software engineering metrics are highly correlated with the class and less correlated amongst themselves. To do that, the CFS technique uses the Pearson coefficient [73], where a high value, or correlation, indicates the best combination of software engineering metrics.

4.3.1.2 **Synthetic Minority Over-sampling Technique (SMOTE)**

SCA tools generate a large number of false positive and false negative warnings, which results in generating datasets that have a disproportionate ratio of true positive, false positive, and false negative warnings, or classes. This problem is known as *unbalanced data* [65]. To solve this problem and enhance the classifier's ability, we used the SMOTE technique [74] to balance the training dataset.

The SMOTE technique balances the binary dataset (in our case: CWE-252-Tool2 and CWE-457-Tool2 see Section 4.4.1) by adjusting the class distribution of a dataset. We apply the SMOTE on our multi-class datasets by following the strategy, proposed by Fernandez et al. [75], in two steps. First, we use the binarization schemes, such as one versus one (OVO), to transform the multi-class dataset into a set of binary datasets.

Second, we apply the SMOTE approach on each binary dataset to solve the imbalance problem. However, using the oversampling technique, such as SMOTE, to balance the datasets may cause overfitting problem. To overcome this problem, we use the cross-validation technique [76].

4.3.1.3 **Classification Techniques**

We selected four kinds of classification techniques for our work: instance-based learning, ensemble learning, rule-based learning, and statistical learning. The techniques are used as benchmarking algorithms to learn and predict the SCA tools warning for a given source code based on its corresponding software engineering metrics.

We also selected ML techniques that have successfully been used in the software defect detection field [77] [78]. We used the Weka Machine Learning workbench (`https://www.cs.waikato.ac.nz/ml/weka/`) and Scikit-learn Machine Learning library (`https://scikit-learn.org/stable/`) to train and test the selected ML techniques. Table 4.1 presents the summary of the four ML techniques we used in the work.

Table 4.1.: Description of ML Techniques Used in Work

ML Technique	**Description**
K-Nearest Neighbor (KNN)	It is an instance-based learning algorithm that generalizes the training data when the time comes to predict a new data point rather than when the training dataset is processed [68].
Support Vector Machine (SVM)	It is developed from statistical learning to build a supervised learning model from either binary datasets or multi-class datasets. In this work, for the multi-class datasets, we use OVO and binary SVM technique to predict the SCA tool warning [67].
Random Forest (RF)	It is an ensemble learning method that constructs a series of unpruned classification trees from bootstrap functions and software engineering metrics of the training dataset. The predicted SCA tool warning is determined using the majority vote as a decision rule [69].
Repeated Incremental Pruning to Produce Error Reduction (RIPPER)	It is a rule induction algorithm that generates the initial set of rules for the minority class using incrementally reduced error. These rules must cover all the functions of that class. Afterward, the algorithm fills up to the next class and repeats the same steps until all the classes have been covered [70].

4.3.2 Selected SCA Tools

We used an open source and a commercial tool for evaluating our work. Since most source code in the Juliet test suite is written in C/C++ and Java, we selected tools that supported at least one of these two languages. The selected SCA tools are listed below [2]:

1. **Tool1.** An open-source SCA tool that uses Lexical analysis to find the flaws in C++ source code. Lexical analysis matches the tokenized source code with a list of checkers, and reports if it finds a suspicious pattern. (See section 3.3.2).

2. **Tool2.** A commercial SCA tool that analyzes both the source code and binaries. To find the flaws in C++ and Java source code, Tool2 builds an abstract model from the source code and then the symbolic execution engine explores the source code to test every execution path and the variables to find the flaws (See section 3.3.2).

We selected an open-source SCA tool because it is freely available and can be used as a base case in order to compare it with a commercial SCA tool. On the other hand, we selected a commercial SCA tool because the commercial tools are usually considered to be more trustworthy than the open-source tools [79].

4.3.3 **Selected Code Base**

We run each SCA tool against the Juliet test suite for C++ to generate the true positive, false positive, and false negative warnings. The NIST Juliet suite contains 61,387 test cases covers 118 CWEs, which aim to create a catalog of software weaknesses and vulnerabilities [55] (see Section 3.3.1).

4.3.4 **Selected Weaknesses (CWEs)**

In this work, we focus on the following CWEs as they have a bigger dataset for SCA tools' warnings when compared to the other weaknesses (see Section 4.4.1) [51], However, the proposed approach is not restricted to a specific number or type of CWEs:

[2]For privacy reasons, we do not disclose the names of the SCA tools.

1. **CWE-252: Unchecked Return Value.** This weakness occurs when the software does not check the return value from the function. This weakness may lead to prevent the software from detecting unexpected states and conditions. To better understand this weakness, Listing 4.1 shows a simplified code snippet that contains a function that prints out the value of **data**.

```
//CWE-252_Unchecked_Return_Value_example
void bad_code(){
        char dataBuffer[100] = "";
        char * data = dataBuffer;
        printLine("Please enter a string: ");
                //POTENTIAL FLAW
        fgets(data, 100, stdin);
        printLine(data);
}
```

Listing 4.1: CWE-252 Example

Without validating the return value of the (**fgets**) function an unchecked return value flaw can be occurred. To avoid this flaw we just need to ensure that the return value of the **fgets** function will be always not Null.

2. **CWE-369: Divide by Zero.** There are two reasons for this weakness; first one is when an unexpected value is provided to the product. The second reason is, if an error occurs that is not properly detected (See section 3.3.3).

3. **CWE-415: Double Free.** This weakness occurs when the product calls **free()** twice on the same memory address. This weakness may lead to modification of unexpected memory locations. To better understand this weakness, Listing 4.2 shows a simplified code snippet that initialize **data** pointer and then freeing the memory.

```
//CWE-415_Double_Free_example
void bad_code(){
    char * data;
    data = NULL;
5         data = (char *)malloc(100*sizeof(char));
6         free(data);
7         /* POTENTIAL FLAW: Possibly freeing memory twice */
8         free(data);
9     }
```

Listing 4.2: CWE-415 Example

There is a potential flaw in line 8 because the **data** pointer may free the memory twice.

4. **CWE-457: Use of Uninitialized Variable.** This weakness occurs when the source code uses a variable that has not been initialized. This weakness may lead to unpredictable or unintended results (See section 3.3.3).

5. **CWE-426: Untrusted Search Path.** This weakness occurs when the software looks out for critical resources using an externally-supplied search path that can point to resources that are not under the application's direct control. To better understand this weakness, Listing 4.3 shows a simplified code snippet that contains a function that use the **POPEN** function, which call opens a process by creating a pipe, forking, and invoking the shell. It does this by executing the command specified by the incoming string function parameter. It creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to write to the pipe.

```
//CWE-426_Untrusted_Search_Path_example
#define BAD_OS_COMMAND "ls -la"
void bad_code(){
        {
            char * data;
            char dataBuffer[100] = "";
            data = dataBuffer;
            if(globalFive==5)
            {
                /* FLAW: the full path is not specified */
                strcpy(data, BAD_OS_COMMAND);
            }
13                 {
14                     FILE *pipe;
15                     /* POTENTIAL FLAW: Executing the popen() function without
                           ↪ specifying the full path to the executable
16                      * can allow an attacker to run their own program */
17                     pipe = POPEN(data, "wb");
18                     if (pipe != NULL)
19                     {
20                         PCLOSE(pipe);
21                     }
22                 }
23             }
24 }
```

Listing 4.3: CWE-426 Example

SCA tool must highlight the potential defects in line 11 and line 17.

6. **CWE-762: Mismatched Memory Management Routines.** This weakness occurs when the application attempts to return a memory resource to the system, but it calls a release function [80].

```
//CWE-762_Mismatched_Memory_Management_Routines_example
void bad_code(char * n){
        char * empname = (char *)calloc(strlen(n) + 1, sizeof(char));
        strcpy(empname, n);
        //POTENTIAL FLAW
        delete empname;
}
```

Listing 4.4: CWE-762 Example

In this example, the function allocates an empname using calloc, however, the function uses delete function to deallocate empname instead of using free(). The SCA tool should be able to detect this defect as a CWE-762 warning message.

7. **CWE-476: NULL Pointer Dereference.** This weakness occurs when the application dereferences a pointer that it expects to be valid, but it is NULL. This weakness may lead to crash or exit (See section 3.3.3).

SCA tools may be able to correctly detect these CWEs, and might also report a set of false positive and false negative warnings, which reduces the usability of the SCA tools. Likewise, going through all the false positive warnings manually, in order to check if the SCA tool correctly detects a real weakness in the source code, will consume a lot of developer time.

4.3.5 **Selected Performance Evaluation Metric**

We have carefully selected a suitable performance measure that examines the strength and the predictive ability of the developed models. SCA tools generate a large number of false positive and false negative warnings and the generated datasets have a disproportionate ratio of the true positive, false positive, and false negative classes. This problem is known as *unbalanced data*. In such a case, when we create a classification model, we will get a high accuracy metric value (such as 90%).

But, this accuracy value is only reflecting the underlying class distribution. This problem is called *accuracy paradox*. For this reason, it is better to avoid using accuracy as the metric to assess the performance of the prediction models [81]. The precision and recall are commonly used as a performance measure in an unbalanced dataset problem [82]. However, there is a trade-off between the precision and recall. Thus it is, therefore, better to use the **F1-score**, which selects the best model based on the balance between the precision and recall, as a performance measure for our comparative needs.

4.3.6 **Selected Software Engineering Metrics**

As indicated earlier, we compute the source code metrics using the Understand tool at the function-level. We do this because we are interested in predicting the behavior of a SCA tool based on how the given function was written not how the whole file was written. For this reason, twenty-one software engineering metrics, which are supported by the Understand tool, are selected to generate the datasets. The selected software engineering metrics are listed below, Section 3.3.4 shows a brief description for these metrics.

1. CountInput (Fan-In).
2. CountOutput (Fan-Out).
3. Knots.
4. CountLineCode.
5. CountLineCodeExe.
6. CountPath.
7. Essential.
8. Cyclomatic.
9. CyclomaticStrict.
10. CyclomaticModified.
11. MaxNesting.
12. MinEssentialKnots.
13. MaxEssentialKnots.
14. RatioCommentToCode.
15. AltCountLineBlank.
16. CountLineBlank (BLOC).
17. CountLineCodeDecl.
18. CountLineComment (CLOC).
19. CountLineInactive.

20. Preprocessor Lines.

21. CountDeclFunction.

However, our proposed approach is not restricted to the selected CWEs or SCA tools. Furthermore, we believe that the proposed approach can work for multiple SCA tools and for different weakness types.

4.4 Experimental Evaluation of SCATWC

This section evaluates the effectiveness of the selected ML techniques. This section also analyzes their performance using the F1-score.

4.4.1 Experimental Setup

In all the experiments, we have adopted a 10-fold cross-validation as a validation method to address the overfitting problem and to obtain a realistic insight about the prediction of the model. In 10-fold cross-validation, the dataset is randomly divided into 10 folds—each one containing the 10% of the data points of the dataset. This means that nine folds were used for training and one fold was used for testing. This procedure is repeated ten times and the final performance value for each ML model is averaged [76].

Table 4.2 summarizes the properties of the generated datasets that we used in our experiments. This table shows the number of data points, the number of features, and the number of classes for each dataset. A majority of the datasets used in this work have either two classes (*i.e.*, true positive and false positive) or three classes (*i.e.*, true positive, false positive, and false negative).

Table 4.2.: Summary Description of the data sets

Data set	**#DataPoints**	**#Features**	**#Classes**
CWE-369-Tool1	1836	21	3
CWE-476-Tool1	805	21	3
CWE-762-Tool1	7277	21	3
CWE-252-Tool2	1263	21	2
CWE-369-Tool2	11186	21	3
CWE-415-Tool2	10388	21	3
CWE-426-Tool2	1064	21	3
CWE-457-Tool2	9165	21	2

4.4.2 **Experimental Results for SCATWC**

4.4.3 **CFS Results Analysis**

For our research, we are interested in building a classifier model that can predict the SCA tool warnings for a given function, but we also interested in finding which of the software engineering metrics are highly correlated with the true positive, false positive, and false negative warnings.

Table 4.3 shows the relevant software engineering metrics that we identified in each dataset after applying CFS to it. Each of these subsets has the highest merit value among 2^{21} other subsets.

To select the best software engineering metrics subset with the highest merit value, the CFS technique uses the Best First Search algorithm [83] to select the software engineering metrics that are highly correlated with the warning type, and they are uncorrelated with each other at the same time.

In this work, to run the CFS technique, we utilized all the features in the generated datasets. In other words, we ran the CFS technique to compute the merit value by investigating all the software engineering metrics that have either positive, negative, or zero correlation values.

On the other hand, we can observe from this table that the most frequent selected software engineering metrics among the eight datasets were CountInput, Knots, CountOutput, CountPath, Cyclomatic, and Essential.

Table 4.3.: Relevant Software Engineering Metrics

Dataset	**Software Engineering Metrics**	**Merit**
CWE-369-Tool1	CountOutput, CountPath, Knots, Cyclomatic, and CountInput	0.45
CWE-476-Tool1	CountInput, CountLineCode, CountPath, and MaxEssentialKnots	0.45
CWE-762-Tool1	CountPath, Knots,CountInput, and MinEssential-Knots	0.73
CWE-252-Tool2	CountInput, Knots, Cyclomatic, CyclomaticStrict, and Essential	0.78
CWE-369-Tool2	CountInput, Knots, CountOutput, CountLineCode, Essential,and CountPath	0.32
CWE-415-Tool2	CountInput, Knots, CountOutput, Essential, and CountPath	0.31
CWE-426-Tool2	CountInput, Knots, Cyclomatic, Essential, Count-Path, and CountDeclFunction	0.40
CWE-457-Tool2	CountInput, Knots, Cyclomatic, and MaxNesting	0.55

4.4.4 **Discussion of Results**

Table 4.4 presents the F1-score for the eight datasets and four classification techniques after using CFS to select the most important software engineering metrics. As shown in this table, we can observe that the RF technique is better than the other ML techniques at predicting the functions that force the SCA tools to emit either the true positive, false positive, or false negative warnings.

For example, for the CWE-369-Tool1 dataset, the RF can correctly predict 94% of the given functions which warnings the SCA tool will emit, while the RIPPER can correctly predict 92% of these functions. Likewise, KNN and SVM predict correctly 87% and 70% what warnings the given function will force the SCA tool to emit.

Table 4.4.: Experimental Results Based on F1-score metric.

Dataset	**SVM**	**KNN**	**Random Forest**	RIPPER
CWE-369-Tool1	70%	87%	**94%**	92%
CWE-476-Tool1	58%	76%	**83%**	81%
CWE-762-Tool1	76%	81%	**87%**	83%
CWE-252-Tool2	84%	85%	**87%**	86%
CWE-369-Tool2	72%	86%	**91%**	87%
CWE-415-Tool2	83%	87%	**94%**	92%
CWE-426-Tool2	67%	83%	**89%**	87%
CWE-457-Tool2	93%	94%	**98%**	95%

The results show that predicted models generated using the RF and RIPPER techniques have F1-score greater than 80% corresponding to most of the datasets. On the other hand, the predicted models that generated using SVM technique have a low value of F1-score among the multi-class datasets. However, SVM technique has a high F1-score (*i.e.*, larger than 80%) among the binary datasets.

The F1-score of the RF models were between 83% - 98% in the eight datasets. The results show that the RF is better than the other ML techniques. It also demonstrates that the RF is the most effective in SCA tool warnings prediction. One reason that the RF technique has better performance is that the RF technique works especially well on large datasets [84] such as CWE-415-Tool2. Another reason is that the CFS selects the optimal subset of software engineering metrics and passes them to RF. This means RF uses the optimal subset of software engineering metrics—giving it a better F1-score score in classifying the SCA tool warnings.

The SVM technique was not able to do well in one dataset of the Tool1 (CWE-476-Tool1), where the F1-score value is only 58%. This is because the SVM technique was not able to make an accurate prediction of the SCA tool warnings on the basis of only the Volume (*e.g.*, CountDeclFunction and CountLineCode) and Object-Oriented metrics (*e.g.*, CountInput).

4.4.5 **RIPPER Results Analysis**

As shown in the experimental results, the RIPPER comes in second place after the RF in achieving high predicting performance. In this section, we display an example of the RIPPER rules learned from the CWE-426-Tool2 dataset, and how we try to interpret these rules. Fig. 4.4 shows sample rules for the Tool2, which are as follows:

1) CountInput >= 2 && CountPath >= 2 $\longrightarrow$ **True-Positive-Alter**
2) CountInput >= 2 && Essential >= 1 && Essential <= 2 $\longrightarrow$ **True-Positive-Alter**
3) CountInput >= 2 && Knots <= 0 $\longrightarrow$ **True-Positive-Alter**
4) CountDeclFun <= 1 && Essential >= 1 $\longrightarrow$ **Fasle-Negative-Alter**
5) Knots >= 0 $\longrightarrow$ **Fasle-Negative-Alter**
6) $\longrightarrow$ **Fasle-Positive-Alter**

Fig. 4.4.: CWE-426-Tool2 Datase Sample of Rules.

1. If the given source code (function) has a Fan-In value larger than or equal 2 (which means that the total number of parameters and global variables that are used in the function is greater than or equal to 2), and also the given function has at least 2 unique paths, then Tool2 can find the existing security flaw in the given function.

2. If the given source code (function) has a Fan-In value larger than or equal 2 (which means that the total number of parameters and global variables that are used in the function is greater than or equal to 2), and also the given function has a complexity larger than or equal to 1 and less than or equal to 2 after all the control-flow structures are replaced with a single statement, then Tool2 can find the existing security flaw in the given function.

3. If the given source code (function) has a Fan-In value larger than or equal 2 (which means that the total number of parameters and global variables that are used in the function is greater than or equal to 2), and also the given function complexity (Knots) equals zero, then Tool2 can find the existing security flaw in the given function.

4. If the number of the function in the given source code equals one and the given source code has a complexity at least 1 after all the control-flow structures are replaced with a single statement, then Tool2 cannot find the existing security flaw in the given function.

5. If the number of the overlapping jumps (*i.e.*, for the corresponding source code, Knots equals to the number of crossing of the lines that determine where every jump in the flow of control occurs) in the given source code(function) larger than zero, then Tool2 cannot find the existing security flaw in the given function.

Lastly, if there exists a data point, or source code, that does not meet the conditions of the previous rules, then the prediction model assigns the majority class in the dataset, which is a false positive warning. In other words, the prediction model assumes that the Tool2 will report that there is a security flaw in the given source code, while in reality there is no one.

From the previous rules, we can conclude that Tool2 can find the defect, or the flaw, in the given function that has a high degree of Fan-In. On the other hand, the ability of the Tool2 in finding the defects in the given function will be reduced when the source code has a high degree of complexity. In this situation, we can infer that the value of the software engineering metrics for the given function affects on the ability of the SCA tool in finding the potential defects in the source code.

Fig. 4.5 shows sample rules for the Tool1, which are as follows:

1. If the given source code (function) has a Fan-In value lower than or equal 2 (which means that the total number of parameters and global variables that are used in function is less than or equal to 2), on the other hand, if the given

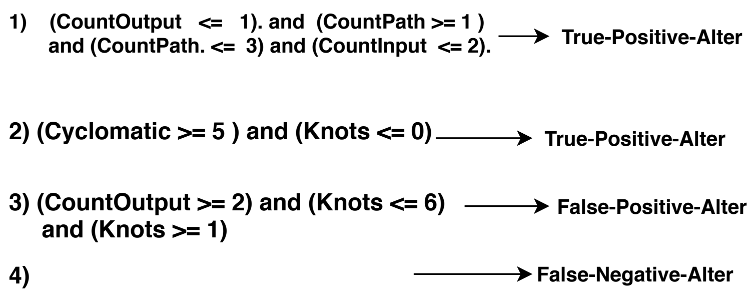

1) (CountOutput <= 1). and (CountPath >= 1) and (CountPath. <= 3) and (CountInput <= 2).
True-Positive-Alter
2) (Cyclomatic >= 5) and (Knots <= 0)
True-Positive-Alter
3) (CountOutput >= 2) and (Knots <= 6) and (Knots >= 1)
False-Positive-Alter
4)
False-Negative-Alter

Fig. 4.5.: CWE-369-Tool1 Dataset Sample of Rules.

function has a Fan-Out value less than or equal to 1. Likewise, if the given function has at most 3 unique paths then the Tool1 can find the existing security flaw in the given function.

2. If the number of crossing of the lines that determine where every jump in the flow of control occurs in the given function equal zero. Also, if the complexity of the given function larger than or equal 5, then Tool1 can find the existing security flaw in the given function.

3. If the given function has a Fan-Out value larger than or equal 2, and the Knots metric value ranges from 1 to 6, then Tool1 will generate a fake warning.

If there exists a data point, or function, that does not meet the conditions of the previous rules the prediction model assigns the majority class in the dataset, which is a false negative warning. In other words, the prediction model assumes that the Tool1 will not report that there is a security flaw in the given source code when in reality there is one.

From the previous rules, we can conclude that Tool1 can find the defect in the given function if it has a low degree of coupling (Fan-In and Fan-Out). On the other hand, Tool1 will generate a false warning when the given function has a high degree of Fan-Out. In this situation, we can infer that the value of the software engineering metrics for the given function affects the ability of the SCA tool in finding the potential defects in the source code. For example, Tool2 cannot highlight the potential defects when the given source code has a high degree of Knots.

Due to a large number of decision trees that the RF technique builds for each dataset (*i.e.*, for the first dataset in Table 4.2 the RF technique creates more than 50 decision trees.), we cannot list and describe the results for each developed model by the RF technique.

4.4.6 **Threat to Validity**

For this work, the threats to validity are related to the software engineering metrics computed by Understand. In most of the software engineering tools, the metrics are computed either at a file or at a function level. We have computed metrics at the function level, which leads results in generating contradictory data points in the datasets.

Another threat to validity is the generalization of the results of the proposed approach. We have analyzed 7,508 test cases from the Juliet test suite for C/C++, which may not truly represent real-world source code.

4.5 **Summary of Contributions**

In this chapter, we have presented the Static Code Analysis Tool Warnings Classification (SCATWC), which is a framework to rank SCA tool warnings using software engineering metrics. The following are the key contributions of the SCATWC.

- Showing how we use ML and data mining techniques along with a collection of software engineering metrics to predict if the source code will lead the SCA tool to emit either true positive, false positive, or false negative warnings; and

- Evaluating which of software engineering metrics are highly correlated with the true positive, false positive, and false negative warnings generated by a SCA tool.

5. RANKING STATIC CODE ANALYSIS (SCA) TOOL WARNINGS

In Chapter 4, we proved that we can use the RF model to classify the generated SCA tools' warnings for Juliet test suite into either true positive, false positive, or false negative. So, in this chapter, we transfer the RF model that trained using Juliet test suite to classify and rank the SCA tools' warnings generated for the real-world source code (open-source software projects). In Section 2.3 of Chapter 2, we have already discussed the related research on ranking SCA tool's warnings. We first describe the challenges associated with ranking SCA tool's warnings generated from the real-world source code in Section 5.1. Next, we present a motivate example in section 5.2. Then, we formally present the Static Code Analysis Tool's Warnings Ranking (SCATWR), which is our novel contribution for ranking warnings and reducing the number of generated fake warnings in efficient way in Section 5.3. In Section 5.5, we describe results of applying SCATWR to different open-source software applications. Finally, we summarize our main contributions in Section 5.6.

5.1 Challenges Addressed by the Proposed Approach

The main limitation of ranking SCA tool warnings is that it is impossible to build and train accurate prediction models for some of the software projects; the software projects either do not have enough developmental historical information (historical data) or have too little historical data. For example, not all software developers maintain a clear list of historical bug information or assemble a set of adequate information from the previous versions of their software. Another example, can be seen when the software developers intend to rank the SCA tool warnings for the first release of a software, which has no historical data. The historical data of the software project was used in current studies to label the generated warnings into true positives or false positives. Then this warnings were used to create a training set to build prediction model.

This model can be used later to predict and rank the future warnings for the new open-source project software release or for the warnings that generated after the current source code was changed to add new functionality or correct some security defect. To address this challenge, plenty of current studies attempt to use the following strategies:

- Gathering warnings for various kinds of open-source software projects to generate a training dataset. The main drawback of this strategy is that it needs human effort to label a large number of warnings and these efforts are too expensive.
- Utilizing another prediction model that trained on another open-source project or synthetic source code to rank the warnings of the target software without retraining the model. Unfortunately, this strategy will harm the predictive performance of the prediction model and may lead to the generation of an inaccurate ranking list of warnings.

To solve these challenges, we proposed the Static Code Analysis Tool's Warnings Ranking (SCATWR) framework, which utilizes one of the most common domain adaptation techniques to rank SCA tool warnings. However, first, we will show a simple example to explain the importance of the proposed framework.

5.2 **Motivating Example**

To demonstrate the importance of reducing false positive warnings, we provide the following concrete example. Table 5.1 summarizes the outputs of one of the SCA tools that run over the Juliet test suite. The output shows the number and percentages of true and false positives reported by the tool.

Table 5.1.: Emitted Warnings for SCA tool on Juliet test suite

CWE-ID	**#TP**	**#FP**	**Warnings**	**Percent of FP**	**Percent of TP**
CWE-126	496	5348	5844	92%	8%
CWE-134	576	16918	17494	96%	4%

From this table, we can observe that 92% and 96% of the emitted warnings were false positives. If we suppose that each warning requires three minutes for manual inspection. The time that the developers need to inspect the emitted warnings for CWE-126 would take 12.18 workdays as to four minutes using our framework. The proposed framework, therefore, can save the developers' time by guiding the software developers toward the most serious warnings only.

5.3 The Approach of SCATWR

This section discusses the design and implementation of the *Static Code Analysis Tool Warnings Ranking (SCATWR)*. The main goal of the SCATWR is to prioritize the warnings reported by the SCA tools for both open-source and synthetic source code using the value of source code metrics computed by the Understand static tool. Figure 5.1 gives an overview of the SCATWR. From this figure, we can observe that the SCATWR framework design is divided into the four main phases, listed below.

5.3.1 Phase #1: Generate Datasets for the Synthetic Source Code

The first phase in the proposed framework was to generate a number of datasets that represent the important characteristics of the functions that forced the SCA tool to emit the warning. Later in this chapter, these datasets were used to train a classifier to rank the warnings as either true or false positives. The input of this phase is the C/C++ Juliet test suite. To generate the datasets we performed the following steps:

1. Collected the SCA tool warnings by running a SCATE framework, which evaluates the quality of the SCA tools in terms of source code metrics [49] [7].

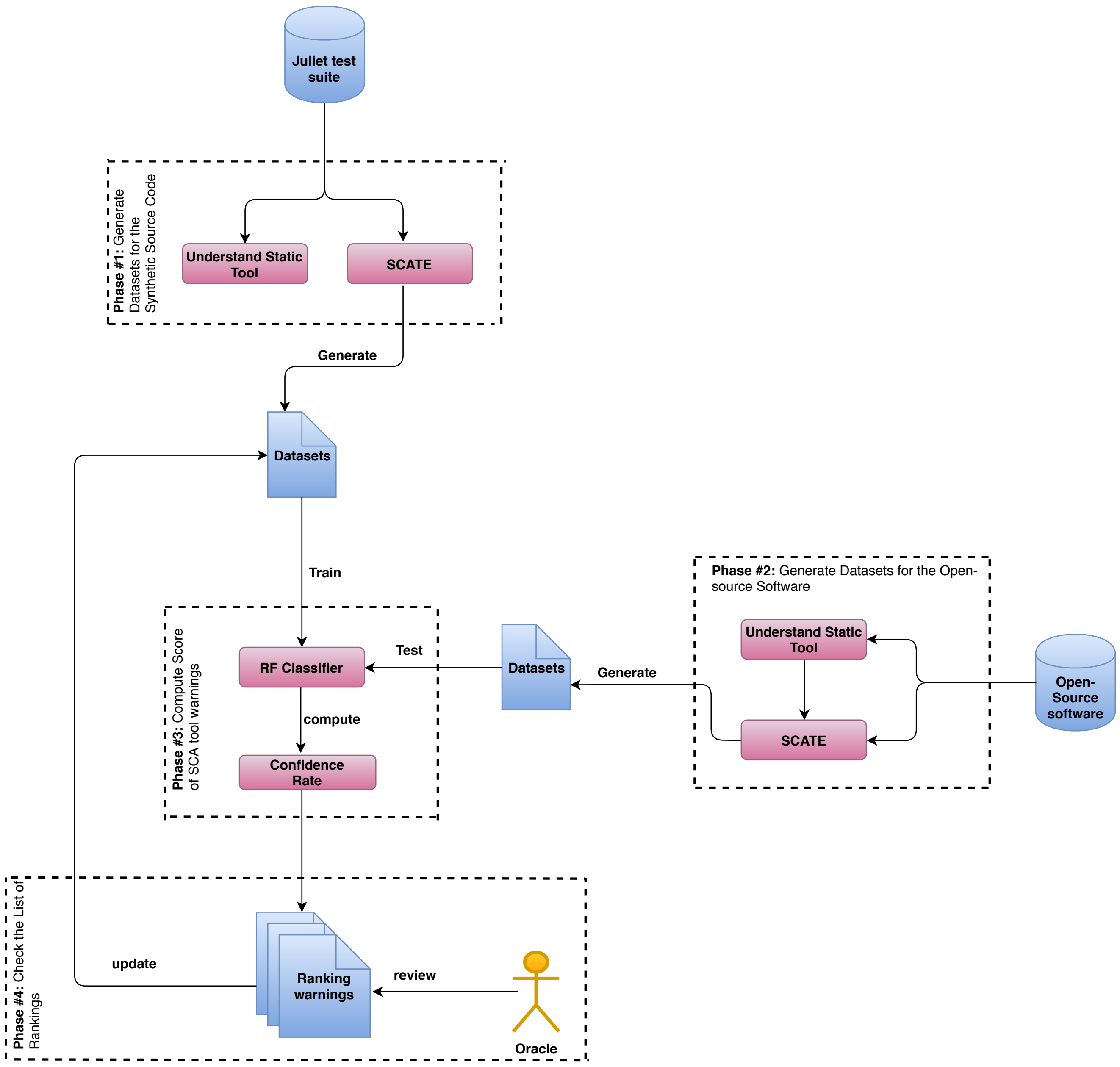


Fig. 5.1.: Overview of SCATWR

2. Computed the value of software engineering metrics for the labeled source code by running the Understand static tool (`https://scitools.com`). Where the value of source code metrics was computed at the function level.
3. Last, created a binary dataset for each CWE. Where the value of the dependent variable (class/label) in the generated dataset is either true or false positive. Furthermore, this dataset was considered as a labeled source domain. For more details see Section 4.2.1

5.3.2 **Phase #2: Generate Datasets for the Open-Source Software**

In this phase, fourteen of open-source software programs were used in order to evaluate the proposed framework. To do that, we created a number of datasets to represent the source code of the open-source software projects that forced the SCA tool to generate the warnings by performing the following steps:

1. Collected the SCA tool warnings by running a SCATE framework after extending this framework to evaluate the SCA tools using some open-source software.
2. Computed the value of software engineering metrics for the labeled source code by running the Understand static tool. As in the previous phase, the value of software engineering metrics was computed at the function level.
3. Last, created an unlabeled dataset for each CWE. This dataset was considered as a target domain.

5.3.3 **Phase #3: Compute Score of the SCA Tool Warnings**

To rank the SCA tool warnings, we first used the RF algorithm to build a strong classifier by creating a forest with a number of weak decision trees. Using the RF algorithm allowed us to overcome the over-fitting problem [85]. In order to train this classifier, we used the generated datasets from **Phase #1**. Next, the trained classifier was used to predict the label value for each warning aggregated from the open-source software (output of **Phase #2**).

However, the output of **Phase #2** (aka target domain), in some experiments that we did later in this work, was randomly divided into two parts (validation set and testing set). The validation set was labeled using the Active Learning (AL) technique, where the instances with the highest entropy value were sent to the professional (oracle) to correctly label them (see Section 5.3.4). Then this validation set was used as labeled target domain to improve the performance of the trained RF classifier. On the other hand, the testing set was used in most of the experiments to evaluate the performance of the proposed approach.

Finally, we used the probabilistic value computed for each warning by using the optimal RF classifier as a confidence rate, in order to rank the warnings as either true or false positive. Based on the confidence rate, the warnings were reordered in a list, where the warnings with the higher confidence rate were at the top of the list and the warnings with the lower confidence rate were at the bottom.

5.3.4 **Phase #4: Check the List of Rankings**

In the last phase of the proposed framework, both the validation set and the ranked list of warnings were sent to professionals to label the validation set correctly and to check the correctness of the ranked list. Based on the professional feedback the training dataset was updated by using the ranked list and the validation set in order to generalize the RF classifier.

In this manner, we mitigated the cost of the manual inspection process for the false positive warnings by forcing the coders and the developers to inspect only the warnings at the top of the ranked list and to ignore the warnings at the bottom. In other words, software developers will omit only the warnings that have a confidence rate below a given threshold.

5.4 Case Study

We use the following case study to evaluate the effectiveness of the proposed framework.

5.4.1 Selected Code Base

In this work, we evaluated the proposed framework using two different kinds of source code as a test cases.

5.4.1.1 NIST Juliet test suite

We ran the SCA tool against the Juliet test suite for C++ to generate the warnings messages. Juliet test suite contains 61,387 test cases covering 118 CWEs and aims to create a catalog of software weaknesses and vulnerabilities [55] [45] (see Section 3.3.1).

5.4.1.2 Open-Source Software

We used fourteen open-source software projects written in C++. As shown in Table 5.2, we selected a wide range of open-source projects. For example, we had projects that we consider small in size (e.g., App2 and App5) and projects we consider large in size (e.g., App7 and App10). We also have projects we consider to be mature by their version number (e.g., App4). Table 5.2 shows some important information about the open-source software projects used to evaluate the proposed framework [1].

[1]For privacy reasons, we do not disclose the names of the open-source software projects

Table 5.2.: Open-source software information

App Name	Version	Language	SLOC	Number of files	Size
App1	3.0.0	C/C++	425682	918	38.6 M
App2	2.14	C/C++	76877	918	7.3M
App3	1.0.1e	C/C++	361381	2203	6.7M
App4	9.2.4	C/C++	650097	5458	38M
App5	1.7.0	C/C++	80676	412	2.3M
App6	1.8.3	C/C++	967716	1728	12M
App7	1.10.2	C/C++	2333668	5109	34M
App8	1.2.2	C/C++	615317	3478	93M
App9	2.8.8	C/C++	736084	6117	35M
App10	1.8.0	C/C++	2538702	3279	32M
App11	1.0.0	C/C++	95676	623	3.4 M
App12	0.26.0	C/C++	80680	502	31.2 M
App13	1.0.0	C/C++	76977	765	70.3 M
App14	3.0.0	C/C++	160195	918	38M

5.4.2 Selected Weaknesses (CWEs)

We used the following Common Weakness Enumerations (CWEs) to investigate the effectiveness of our framework.

1. **CWE-126: Buffer Over-Read.** This weakness occurs when the pointer or its index is incremented to a position beyond the bounds of the buffer or when pointer arithmetic results in a position outside of the valid memory location to name a few. This may result in exposure of sensitive information or possibly a crash [86]. To better understand this weakness, Listing 5.1 shows a simplified code snippet that contains a function that print out the value of the **dest** variable.

```
//CWE-126_Buffer_Overread_example
#include <wchar.h>

void bad_code(){
    wchar_t data[150], dest[100];
    /* Initialize data */
    wmemset(data, L'A', 149);
    data[149] = L'\0';
    wcsncpy(dest, data, 99);
    /* Potential FLAW: do not explicitly null
        terminate dest after the use of wcsncpy() */
    printWLine(dest);
}
```

Listing 5.1: CWE-126 Example

In this example, since the number of characters that being copied to the **dest** is lower than the size of the **data**; **wcsncpy** function will not explicitly null terminal **dest**. So, SCA tool should be able to identify the potential flaw that may occur when using **printWLine** function to print out the value of **dest** as CWE-126 warning message.

2. **CWE-134: Use of Externally-Controlled Format String.** This weakness occurs when the function has a format string as one of its arguments, and this format string constructs from an external source. This may result in denial of service or data representation problems [87]. To better understand this weakness, Listing 5.2

```
//CWE-134_Use_of_Externally_Controlled_Format_String_example
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main (int argc, char **argv){
        char buf [100];
        int x = 1 ;
        snprintf ( buf, sizeof buf, argv [1] ) ;
        buf [ sizeof buf -1 ] = 0;
        //Potential Flaw
        printf ( "Buffer size is: (%d) \nData input: %s \n" , strlen (buf)
            ↪ , buf ) ;
        printf ( "X equals: %d/ in hex: %#x\nMemory address for x: (%p) \n"
            ↪  , x, x, &x) ;
}
```

Listing 5.2: CWE-134 Example

> Now, If the code snippet received the string "John %x %x" as input the format function **snprintf** parses the input string and the output will be the name John and the contents of the memory address [88].

We selected these CWEs because they have a bigger dataset for SCA tool warnings when compared to the other weaknesses (see Section 5.5.2). Although, we are using the CWEs mentioned above for our work, the proposed framework is not limited to a specific number or type of CWE. The challenge for us, however, is finding datasets that contain enough warnings for a CWE to validate the proposed framework.

5.4.3 **Selected Static Code Analysis (SCA) Tools**

We used one open-source SCA tool (Tool6) to evaluate our work. Tool6 supports C/C++ programming language. The selected SCA tool is a quite simple tool. This tool examines C/C++ source code and reports any possible security vulnerabilities sorted by risk level.

Tool6 uncovers many common software weaknesses, such as buffer overflow risks, format string problems, and race conditions, using a built-in database of dangerous C/C++ constructions. We selected an open-source tool because it is freely available. On the other hand, we chose Tool6 from a long list of open-source tools because it supports the most common software weaknesses, such as CWEs.

5.4.4 **Selected Software Engineering Metrics**

To rank SCA tool warnings, 21 of source code metrics were computed at a function level for the given source code. These metrics were categorized into three main groups. Table 5.3 shows a brief description of these groups for more information see Section 3.3.4 and 4.3.6.

Table 5.3.: Description of Source Code Metrics [62]

Source Code Metric	**Description**
Complexity Metrics	Compute the complexity of a given function. For example, Knots metric reflects the structural complexity of a given source code by measuring the overlapping jumps. From this group we used Knots, Essential, Cyclomatic, CyclomaticStrict, CyclomaticModified, MaxNesting, CountPath, MaxEssentialKnots, and MinEssentialKnots.
Volume Metrics	Reflect the number of line in a given source code that satisfies some conditions. For example, CountLineCode metric reflects the total number of lines that contain source code only in a given function. From this group we used CountLineCode, CountLineCodeExe, RatioCommentToCode, AltCountLineBlank, CountLineBlank, CountLineCodeDecl, CountLineComment, CountLineInactive, Preprocessor Lines, and CountDeclFunction.
Object-Oriented Metrics	Compute the coupling for a given function. For example, CountInput metric Computes the Fan-In for given source code. From this group we used CountInput and CountOutput.

5.4.5 **Selected Machine Learning (ML) Technique**

In Chapter 4, we demonstrated that we can use source code metrics to classify warnings generated by two SCA tools (open-source and commercial tools) as a true positive, false positive, or false negative warnings.

Likewise, the experimental results show that the classifier generated by RF technique outperformed the other classifiers generated by the other ML techniques. For this reason, we decided to use RF to rank the SCA tool warnings as either true positive or false positive. The RF classifier is an ensemble classifier that produces multiple decision trees using a randomly selected subset of training samples and variables [85].

5.4.6 **Selected Performance Metric**

We used F1-score [89] to evaluate the performance of the classifier used in our framework [90]. We selected F1-score because the generated datasets from both NIST Juliet test suite and the open-source software suffer from an imbalance issue.

We, therefore, cannot use the accuracy metric, which measures the number of correct predictions made by the model over all kinds of predictions made [90], to measure classifier performance. F1-score provides the weighted average of the precision and recall—where an F1-score reaches its best value at 1 and the worst value at 0. The relative contribution of precision and recall to the F1-score are equal. F1-score is given with the following equation.

$$F1 - score = 2 * \frac{(precision * recall)}{(precision + recall)} \tag{5.1}$$

5.4.7 **Selected Domain Adaptation (DA) Technique**

Traditional ML techniques work well when both the training and testing datasets are drawn from the same feature space and the same distribution. When the distribution changes, most statistical methods need to be rebuilt from scratch using newly collected training data.

To solve this problem Transform Learning (TL) suggests applying the knowledge that learned previously(from the source domain) to solve new problems (target domain) faster or with better solutions [91]. In the following subsections, we briefly review some basics of notations and concepts that are usually used in the domain adaptation field.

5.4.7.1 **Problem Settings**

Given the labeled functions in the source domain as $D_S = \{(x_1^s, y_1^s), ..., (x_{ns}^s, y_{ns}^s\} = \{X_S, y_S\}$ and the unlabeled functions in the target domain as $D_{T,U} = \{(x_1^{t,u}), ..., (x_{nt,u}^{t,u}\} = \{X_{T,U}\}$. Sometimes, we may also have a small amount of labeled data from the target domain as $D_{T,L} = \{(x_1^{t,l}, y_1^{t,l}), ..., (x_{nt,l}^{t,l}, y_{nt,l}^{t,l}\} = \{X_{T,L}, y_{T,L}\}$. Moreover, the entries in y_S and $y_{T,L}$ denote their corresponding labels(1 (true positive warning) and 2(false positive warning)).

Therefore, our proposed framework aims at predicting the label value for the unlabeled target domain using the source domain and the labeled data from the target domain by building the optimal model that minimizes the expected loss with respect to the true distribution $P(X, Y)$.

In general, there is two main distributional difference between the source and target domains, namely, Instance difference and labeling difference. The difference between the source and target domains may come from the difference between the marginal distribution. In another word, $P_s(X) \neq P_t(X)$ but $P_s(Y|X) = P_t(Y|X)$. This problem can be referred to as a covariate shift [92] [93] or sample selection bias [94].

On the other hand, the difference between the source and target domains may come from the difference between the conditional probability distribution. In another word, $P_s(Y|X) \neq P_t(Y|X)$ but $P_s(X) = P_t(X)$. This problem can be referred to as labeling difference. In order to discover if our datasets (Juliet test suite and open-source software projects) have labeling difference, we need first to label some of instances from the target domain.

If label the target domain instances is a time-consuming and costly process in this situation we have to assume that there is no labeling difference between the source and target domains. To solve the above problems we used instance weighting technique [95] [96]. The main goal of this technique is to assign instance-dependent weights to the loss function when minimizing the expected loss over the data distribution [95] [96].

To find the optimal prediction model for target domain from a pool of models we should find the model that will minimize the expected loss with respect to the joint distribution $P(X,Y)$.

$$f^* = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} P(x,y) L(x,y,f) \tag{5.2}$$

Where f^* is the optimal model and $L(x,y,f)$ is a loss function. To find the optimal model for the target domain (open-source software projects), we follow the approach proposed by Jiang [96].

5.4.7.2 **Instance Weighting Technique**

Instance-based transfer learning techniques is considered as one of the common TL technique that can be used to solve the domain adaptation problem. Instance-based transfer learning assumes that certain parts of the data in the source domain can be reused for learning in the target domain by re-weighting.

There are two major techniques can be used to re-weight source domain instances: instance re-weighting and importance sampling. In this research we used the framework proposed by Jiang [97] to remove the misleading warnings in the Juliet test suite datasets, re-weight the Juliet test suite warnings to simulate the unlabeled open-source software projects' warnings, and finally, use a small set of labeled warnings from the open-source software project's warnings to improve the predictive capability of the ML model.

The proposed weighting framework suggest using three datasets: labeled source (D_s), labeled target ($D_{t,l}$), and unlabeled target domains ($D_{t,u}$). First source domain instances were used to approximate the expected loss in the target domain using the following formula [97].

$$f_t^* = \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} \frac{P_t(x,y)}{P_s(x,y)} P_s(x,y) L(x,y,f) \tag{5.3}$$

$$\approx \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_s} \frac{P_t(x_i^s)}{P_s(x_i^s)} \frac{P_t(y_i^s|x_i^s)}{P_s(y_i^s|x_i^s)} L(x_i^s, y_i^s, f) \tag{5.4}$$

$$= \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_s} \alpha_i \beta_i L(x_i^s, y_i^s, f) \tag{5.5}$$

Second, the labeled target domain instances only were utilized to approximate the expected loss in the target domain using the following formula.

$$f_t^* = \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} P_t(x,y) L(x,y,f) \tag{5.6}$$

$$\approx \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} \widetilde{P_t}(x,y) L(x,y,f) \tag{5.7}$$

$$= \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_{t,l}} L(x_i^{t,l}, y_i^{t,l}, f) \tag{5.8}$$

Third, the unlabeled target domain instances could be used to find the optimal learning model for the target domain using the following formula.

$$f_t^* = \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} P_t(x) P_t(y|x) L(x,y,f) \tag{5.9}$$

$$\approx \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} \widetilde{P_t}(x) P_t(y|x) L(x,y,f) \tag{5.10}$$

$$= \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_{t,u}} \sum_{y\in\mathcal{Y}} P_t(y|x_i^{t,u}) L(x_i^{t,u}, y, f) \tag{5.11}$$

$$= \underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_{t,u}} \sum_{y\in\mathcal{Y}} \gamma_i(y) L(x_i^{t,u}, y, f) \tag{5.12}$$

Finally, all the previous formulas were combined into a single objective function. As shown below.

$$\begin{aligned} f_t = \underset{f\in\mathcal{H}}{\operatorname{argmin}} \Bigg[& \lambda_s \sum_{i=1}^{N_s} \alpha_i \beta_i L(x_i^s, y_i^s, f) + \lambda_{t,l} \sum_{i=1}^{N_{t,l}} L(x_i^{t,l}, y_i^{t,l}, f) \\ & + \lambda_{t,u} \sum_{i=1}^{N_{t,u}} \sum_{y\in\mathcal{Y}} \gamma_i(y) L(x_i^{t,u}, y, f) + \lambda R(f) \Bigg] \end{aligned} \tag{5.13}$$

The last formula (5.13) was employed in this work in order to approximately find the optimal learning model for the target domain. We selected this technique because it has successfully been used in the Natural Processing Language (NPL) field [98].

5.5 Experimental Evaluation of SCATWR

In this section, we present and discuss the experimental results obtained with the proposed framework.

5.5.1 Experimental Setup

To setup and execute our experiment, we executed the following steps:

1. We selected the SCA tool that involves CWEs in its rules.
2. We selected the code base to generate SCA tool warnings. The code base includes Juliet test suite and open-source software.
3. We ran SCA tool against the source code in both the Juliet test suite and the open-source software. We then capture the emitted warnings of the SCA tool because we need this to generate a number of datasets.

4. We assessed the value of the software engineering metrics for each CWE by executing Understand static tool against the source code for the corresponding test cases in Juliet and open-source software.
5. We used scikit-learn, which is a free software machine learning library for the Python programming language to train the RF classifier over the emitted datasets from the Juliet test suite.
6. We classified and ranked the open-source software warnings as true or false positives using the trained RF classifier after transfer it using instance weighting technique;
7. We computed the threshold for the false positive warnings for each CWE.
8. We created the warnings ranked list by comparing the probability value for each warning with the threshold.
9. Last, we asked the professional to review and check the ranked list of warnings.

5.5.2 **Dataset Statistics**

To evaluate the proposed framework we generate four of datasets from the Juliet test suite and open-source software projects. Table 5.4 summarizes the properties of the emitted datasets that we used in our experiments. This table shows the number of data points (warnings), the number of features (source code metrics), and the number of classes for each dataset. A majority of the datasets used in this chapter have two classes (*i.e.*, true positive and false positive).

Table 5.4.: Summary Description of the datasets

Dataset Name	**Data Points**	**Features**	**Classes**
CWE-126-Tool6-open-source	1200	21	2
CWE-134-Tool6-open-source	351	21	2
CWE-126-Tool6-Juliet	5844	21	2
CWE-134-Tool6-Juliet	17494	21	2

5.5.3 **Experimental Results**

Since the false positive warnings reduction approach proposed in this work heavily depends on the confidence value computed by the RF Classifier, we perform a set of experiments to select the optimal classifier, which will transfer the knowledge from the source domain (Juliet test suite) to the target domain (open-source software projects). The setting and result for each experiment are listed below.

5.5.3.1 **Experiment #1: Using a Source Baseline Model for Ranking**

In Chapter 4, we demonstrated that the RF classifier is well suited for predicting and classifying SCA tool warnings across a wide range of synthetic test cases. In this experiment, an RF classifier has been trained using only the warnings aggregated from the Juliet test suite to accurately rank the open-world software warnings. This classifier will be considered as a source baseline model for our work.

Figure 5.2 showcases the steps that we followed to train and test the effectiveness of the source baseline model. This figure depicts our attempt to explore the efficiency of the source baseline model by directly applying the supervised learning model (RF) trained using D_s without weighing the source domain instances for the target domain.

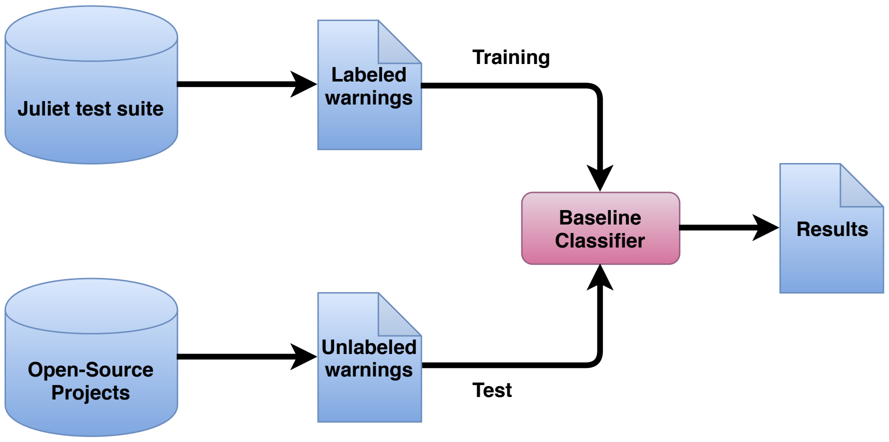


Fig. 5.2.: Using Source Baseline Model to Rank Open-World Software Warnings

In other words, we used this model to directly obtain a classifier for the target domain utilizing the following formula.

$$f_t = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_s} L(x_i^s, y_i^s, f) \tag{5.14}$$

By analyzing the source baseline model results given in Figure 5.3, we can make the following observations:

- The source baseline model is not appropriate to predict or rank the warnings of the open-source software. This can be inferred from the value of the F1-score for the CWE-134-open-source and CWE-126-open-source datasets.

- Since the Juliet test suite is a synthetic source code, the source code will be relatively simple compared to the source code of the open-source software projects. In other words, the number and type of the loops, control structure and function calls used in the open-source software projects differ from those used in the Juliet test suite. This difference causes the software engineering metrics to have a varying range of values that may be larger or smaller than Juliet test suite metrics. Likewise, this difference leads the proposed framework to generate

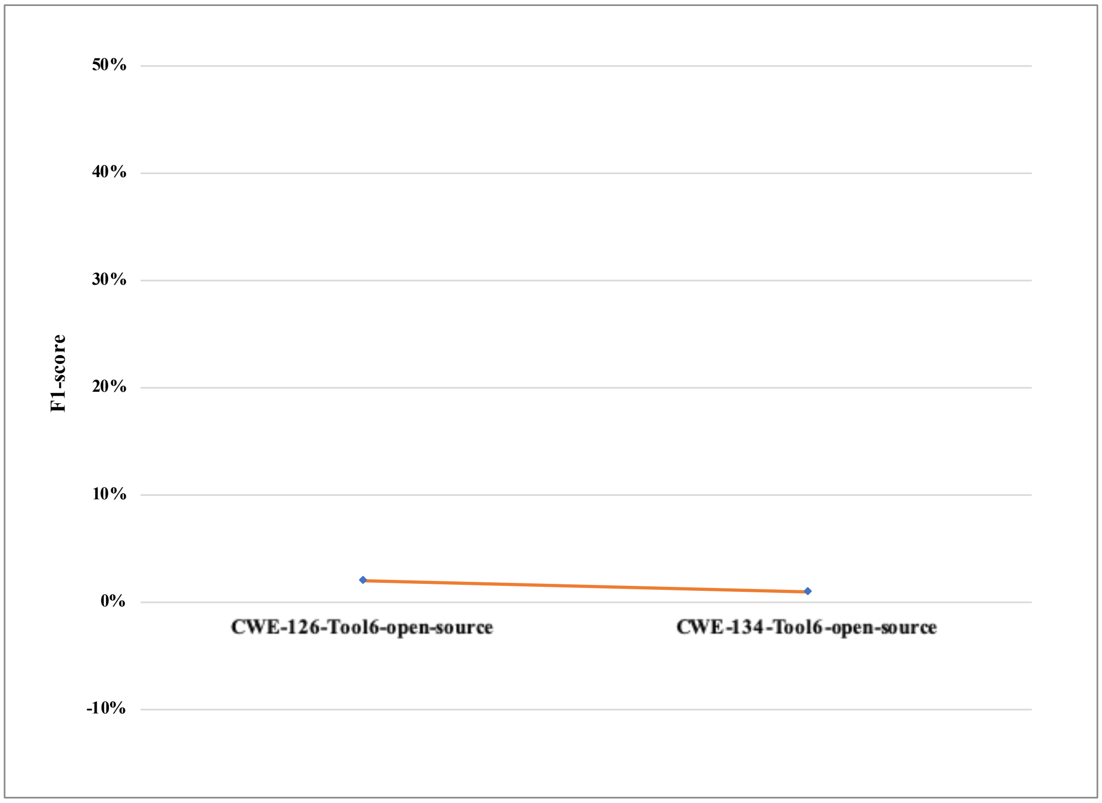


Fig. 5.3.: Source Baseline Model Results

training and testing datasets that are drawn from different distributions, which decreases the performance of the source baseline model.

From these observations, we can conclude that the source baseline model will not be the optimal classifier to rank the SCA tool warnings. Therefore, we need to perform another experiment with different settings and datasets.

5.5.3.2 **Experiment #2: Using a Target Baseline Model for Ranking**

As seen in the previous experiment, using only the Juliet test suite warnings only to train the RF classifier and then directly predict the open-source software project warnings led the classifier to perform badly. So, in this experiment, instead of using Juliet test suite warnings to train the classifier, we used a subset of the open-source software project warnings after labeling them using the AL technique. Therefore, formula (5.6) was used in this experiment.

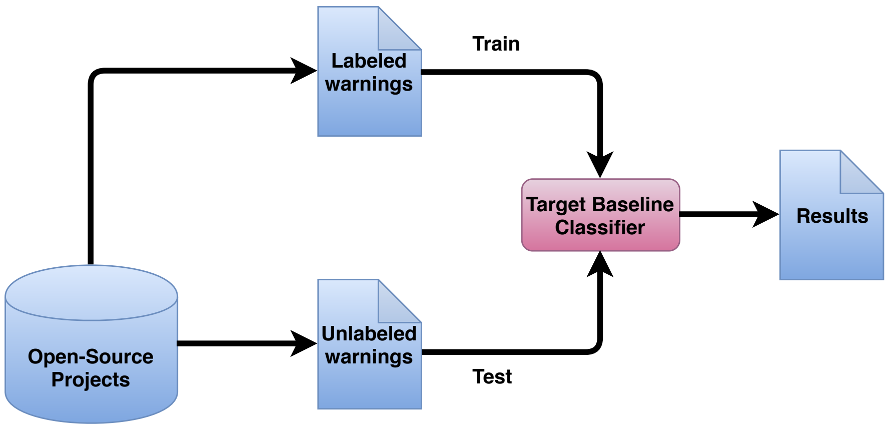


Fig. 5.4.: Using a Target Baseline Model to Rank Open-World Software Warnings

Figure 5.4 showcases the steps that we followed to train and test the effectiveness of the target baseline model. This figure depicts our attempts to explore the efficiency of the target baseline model by directly applying the supervised learning model (RF) trained using $D_{t,l}$ without considering the source domain instances (Juliet test suite warnings) for the unlabeled target domain (open-source software project warnings of unknown type).

However, by analyzing the target baseline model results that are given in Figures 5.5 and 5.6, we can make the following observations:

- The target baseline model is an appropriate to predict or rank the warnings of the open-source software. This can be inferred from the value of the F1-score for CWE-134-Tool6-open-source and CWE-126-Tool6-open-source datasets.

- Increasing the number of labeled warnings from the target domain to train the model can improve the performance of the model and enhance the correctness of the open-source warnings list. This is because more warnings in the validation set or the labeled target domain mean that the approximate distribution will be better.

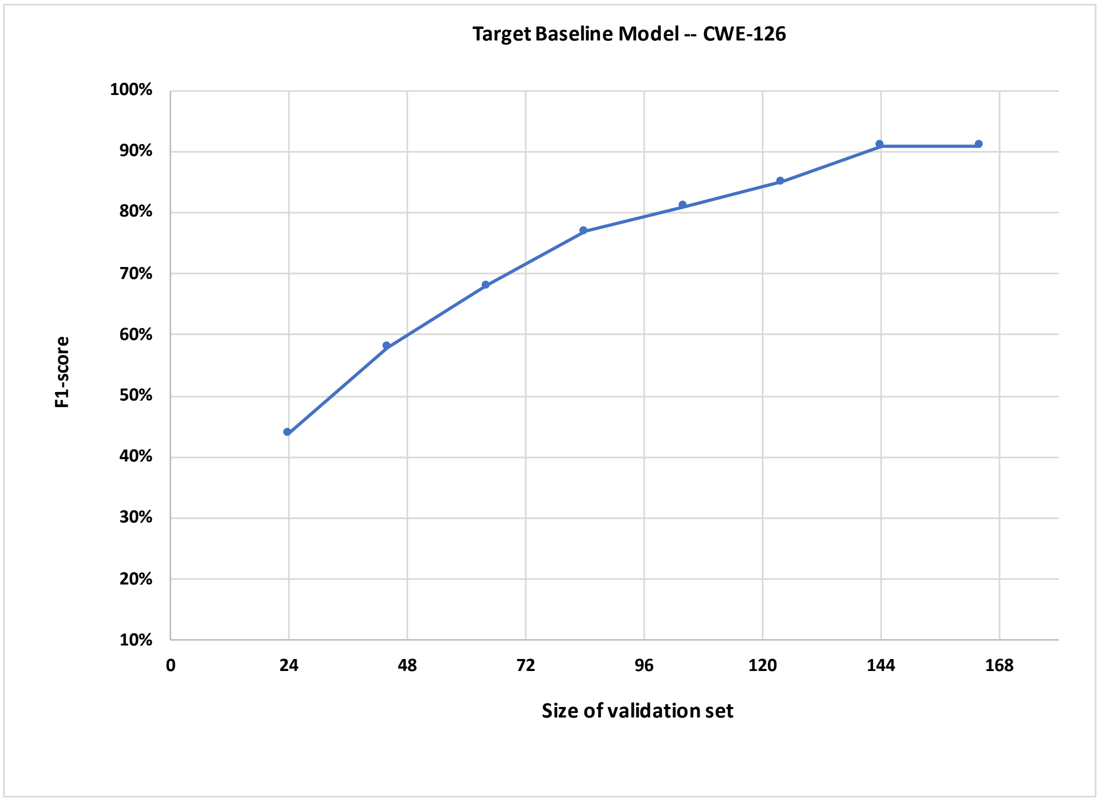


Fig. 5.5.: Target Baseline Model Results
for CWE-126

From these observations, we can conclude that the target baseline model can be an optimal classifier to rank SCA tool warnings if we have enough labeled warnings from the open-source software projects to start the AL technique. Unfortunately, as we mentioned before, we cannot depend solely on the labeled target domain to train the classifier because some software projects do not have any historical data or bug lists to build a training dataset. On the other hand, depending on humans to label some target domain warnings may generate training datasets with mislabeled warnings. Therefore, we need to perform another experiment to explore whether the

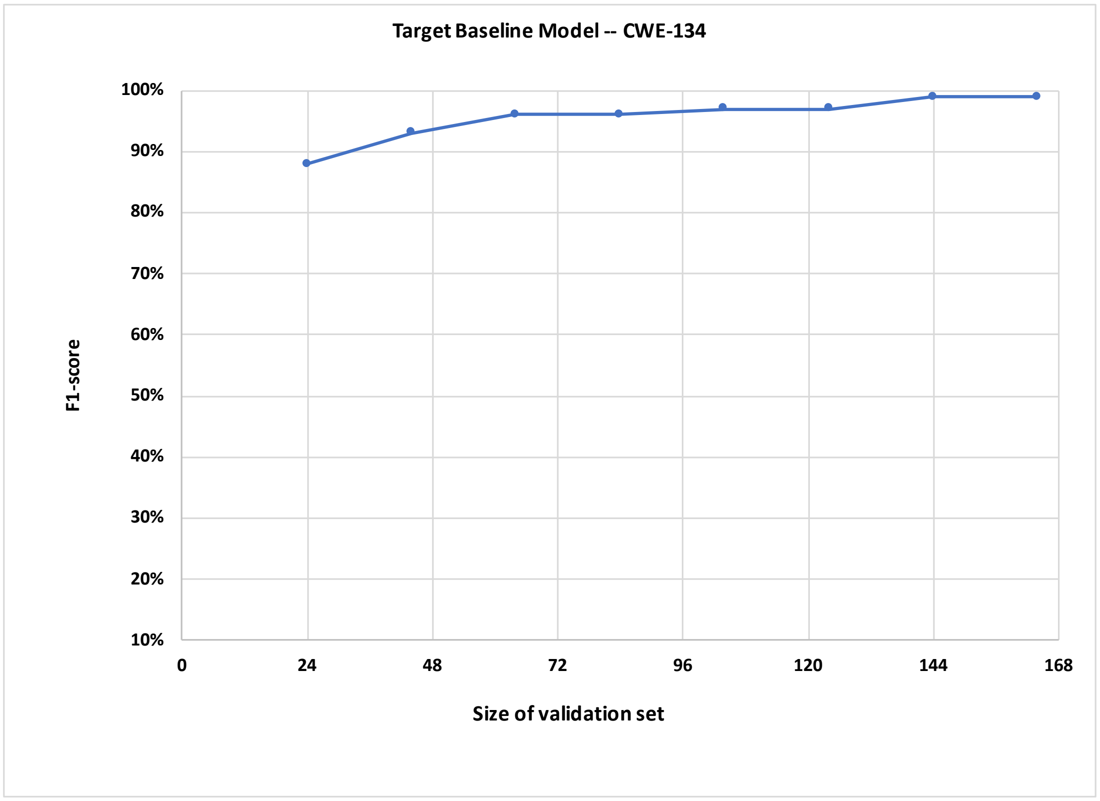


Fig. 5.6.: Target Baseline Model Results
for CWE-134

performance of the classifier will be improved when we use both the labeled warnings from the open-source software projects and the warnings from the Juliet test suite.

5.5.3.3 Experiment #3: Using a Baseline Model Trained Over D_s and $D_{t,l}$

Figure 5.7 displays the steps that we followed in this experiment to train the RF classifier and test it with the unlabeled open-source software projects. From this figure, we can observe that we did not reweigh any Juliet test suite warnings to train the model. In other words, all the Juliet test suite warnings equally collaborated to extract the important pattern from the source code of the Juliet test suite, such as the number of unique paths that the source code could use to achieve specific tasks.

Likewise, we utilized some of the labeled open-source warnings to train the model. In this experiment we utilized the following formula to build the RF classifier.

$$f_t = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_{s+t,l}} L(x_i, y_i, f) \tag{5.15}$$

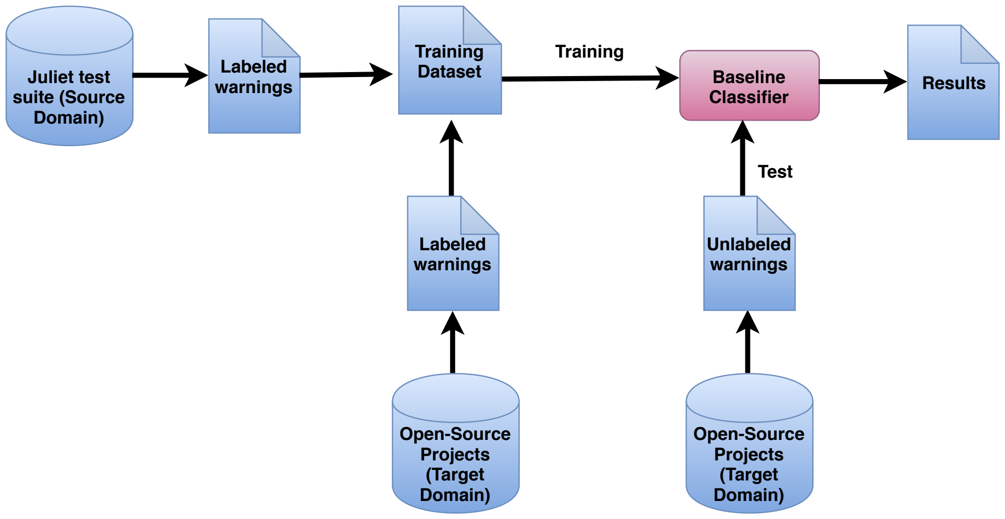

Juliet test suite (Source Domain)
Labeled warnings
Training Dataset
Training
Baseline Classifier
Results
Test
Labeled warnings
Unlabeled warnings
Open-Source Projects (Target Domain)
Open-Source Projects (Target Domain)

Fig. 5.7.: Using a Baseline Model Trained Over D_s and $D_{t,l}$ to Rank Open-Source Software Warnings

However, by analyzing the model results that are given in Figures 5.8 and 5.9, we can make the following observations:

- The RF classifier trained using both the SCA tool warnings that aggregated from a synthetic source code and the warnings collected from some open-source software projects is appropriate to predict or rank the rest of the warnings from the other open-source software projects. This can be inferred from the values of the F1-score for the CWE-134-Tool6-open-source and CWE-126-Tool6-open-source datasets.

- Using a labeled target domain to train the RF classifier helps the model to learn new patterns appearing in the target domain warnings (open-source software projects) only. These patterns do not exist in the dataset that generated using

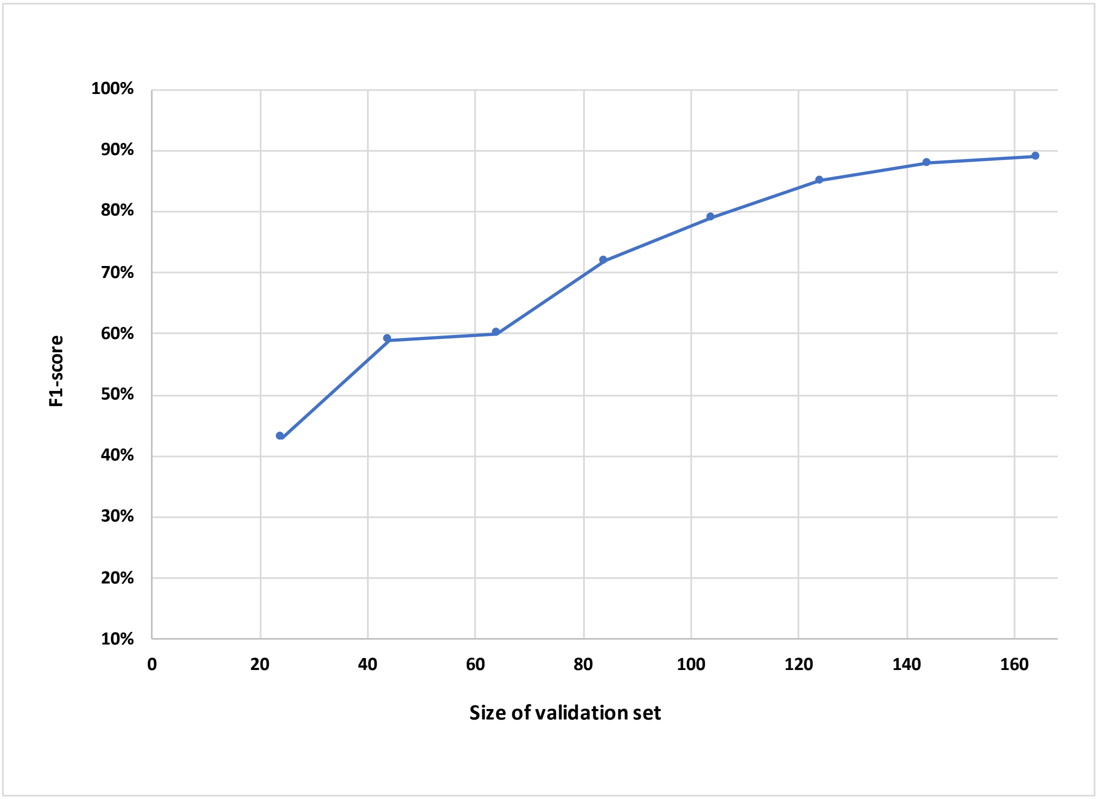


Fig. 5.8.: Result of Baseline Model Trained
Over D_s and $D_{t,l}$ for CWE-126

the Juliet test suite, as the real-world source code structure, which could increase the complexity of the source code and make it difficult for SCA tool to understand and analyze the source code to determine the potential defects.

- Again, increasing the number of labeled warnings from the target domain to train the model can improve the performance of the model and enhance the correctness of the open-source warnings list. This is because more warnings in the validation set or labeled target domain means that the approximate distribution will be better.
- Last, using warnings from synthetic source code such as Juliet test suite can reduce the likelihood of having some of the mislabeling warnings in the training dataset, because usually synthetic test cases were built to evaluate the SCA tools

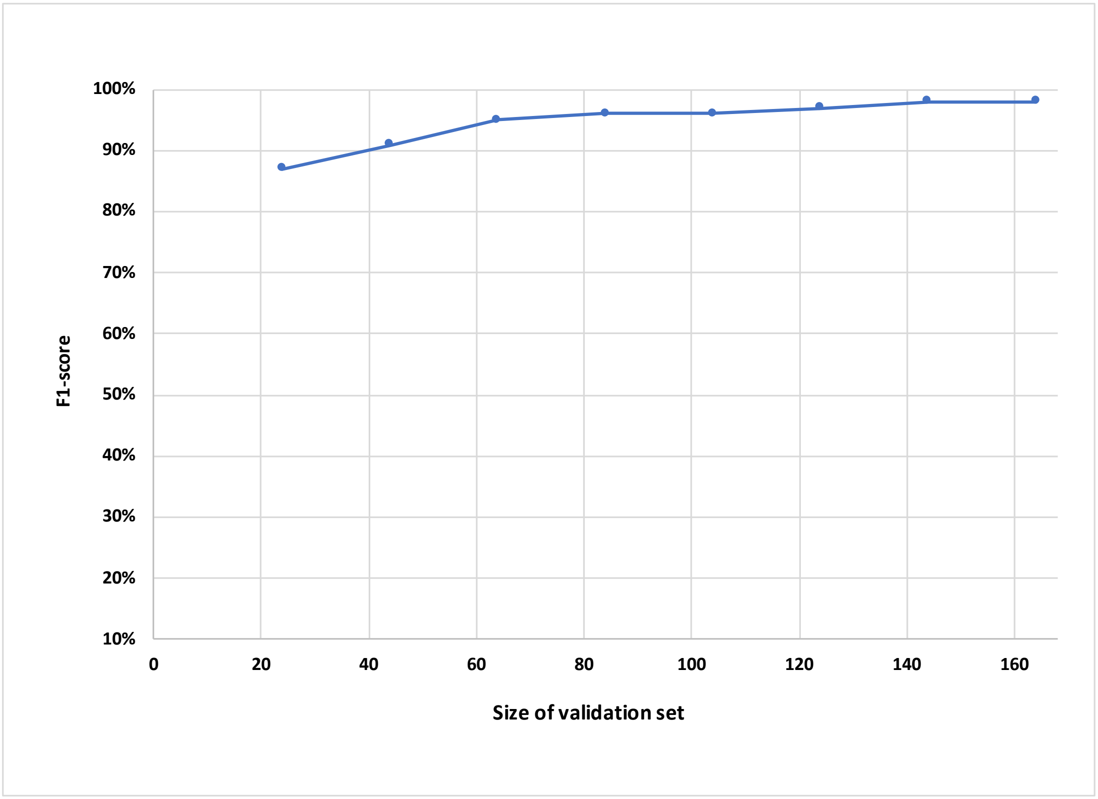


Fig. 5.9.: Result of Baseline Model Trained Over D_s and $D_{t,l}$ for CWE-134

by adding some annotation to the source code in order to determine the type and the location of the potential defects in the source code. These annotations help us to correctly label the SCA tool warnings, which improve the performance of the classifier.

Building the RF classifier in this experiment depends on the availability of professional or a large number of open-source software labeled warnings. However, in some domains, it is very difficult to construct a large labeled dataset due to the costly annotation. This can be considered as a major limitation for this experiment, which the next experiment was intended to overcome.

5.5.3.4 **Experiment #4: Baseline Model Trained Using α and Juliet Suite**

Figure 5.10 summarizes the steps that we followed in this experiment to train the RF classifier using the Juliet test suite warnings reweighed by α value. In this experiment, we used the α to reduce the feature distribution divergence between the Juliet test suite and open-source software warnings.

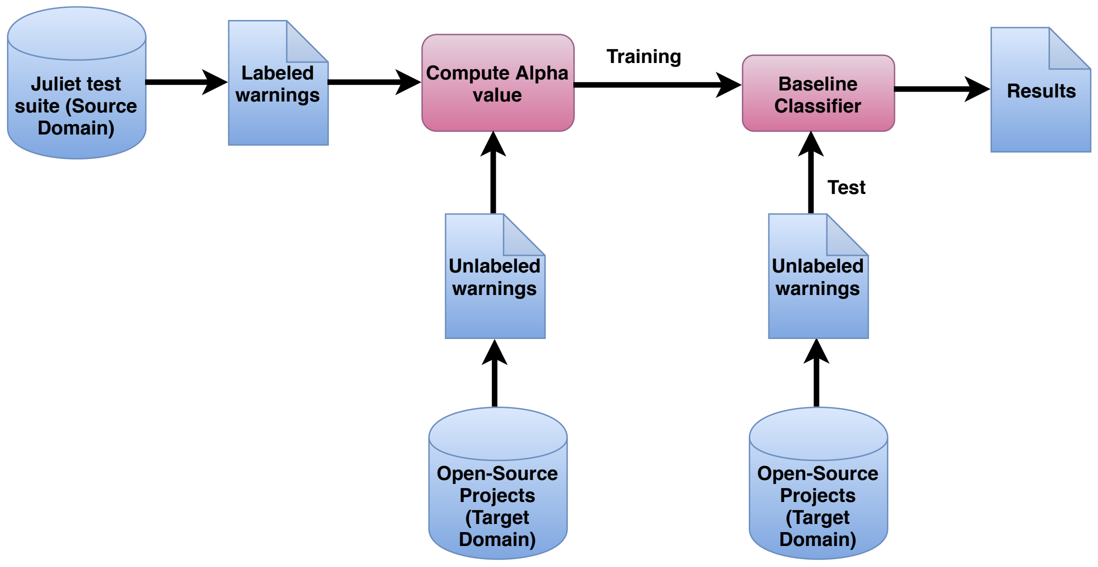


Fig. 5.10.: Using a Baseline Model Trained Using α and Juliet Test Suite to Rank Open-Source Software Warnings

To compute the value of α, we used both the unlabeled open-source software projects and the logistic regression method proposed by Zadrozny [94] by rewriting $\frac{P_t(x)}{P_s(x)}$ as follows.

$$\begin{aligned}
\frac{P_t(x)}{P_s(x)} &= \frac{P(x|d=t)}{P(x|d=s)} \\
&= \frac{P(d=t|x)P(x)}{P(d=t)} . \frac{P(d=s)}{P(d=s|x)P(x)} \\
&= \frac{P(d=s)}{P(d=t)} . \frac{P(d=t|x)}{P(d=s|x)} \\
&\propto \frac{P(d=t|x)}{P(d=s|x)}
\end{aligned} \tag{5.16}$$

Where d denotes either the source (Juliet test suite) or target (open-source software projects) domain. To find the value of $P(d=t|x)$ and of the $P(d=s|x)$, we built a logistic regression model as follows.

$$\begin{aligned}
P(d=t|x) &= \frac{1}{1+exp(-\theta^T \mathbf{x})} \\
P(d=s|x) &= 1 - P(d=t|x) \\
&= \frac{1}{1+exp(\theta^T \mathbf{x})}
\end{aligned} \tag{5.17}$$

To learn the logistic regression (θ), we considered the Juliet test suite warnings as belonging to one class and the open-source software warnings as belonging to the other class. The value of α_i was be computed as follows [97].

$$\begin{aligned}\alpha_i' &= \frac{P(d=t|x_i^s;\theta)}{P(d=s|x_i^s;\theta)}\\ &= \frac{1+exp(\theta^T\mathbf{x}_i^s)}{1+exp(-\theta^T\mathbf{x}_i^s)}\\ C &= \sum_{i=1}^{N_s}\alpha_i'\\ \alpha_i &= \frac{N_s}{C}\alpha_i'\end{aligned} \tag{5.18}$$

On the other hand, we set $\lambda_s = 1$, $\lambda_{t,l} = 0$,and $\lambda_{t,u} = 0$. The final formula that we used in this experiment is listed below.

$$f_t^* = \underset{f\in\mathcal{H}}{\operatorname{argmin}}\sum_{i=1}^{N_s}\alpha_i L(x_i^s, y_i^s, f) \tag{5.19}$$

However, by analyzing the model results given in Figure 5.11, we can make the following observations:

- The RF classifier trained using only the SCA tool warnings that aggregated from a synthetic source code, having the highest weight and ignoring the Juliet test suite warnings of low weight outperform the baseline model trained using only Juliet test suite warnings. Unfortunately, F1-score was lower than 20%. Thus, we cannot use this model to reduce the SCA tool's false positive warnings. This can be inferred from the values of the F1-score for CWE-134-Tool6-open-source and CWE-126-Tool6-open-source datasets.

- From our experimental results, we can conclude that the Juliet test suite warnings may have few or no labeled warnings representing the open-source software warnings located in the dense regions. This decreases the RF classifier performance.

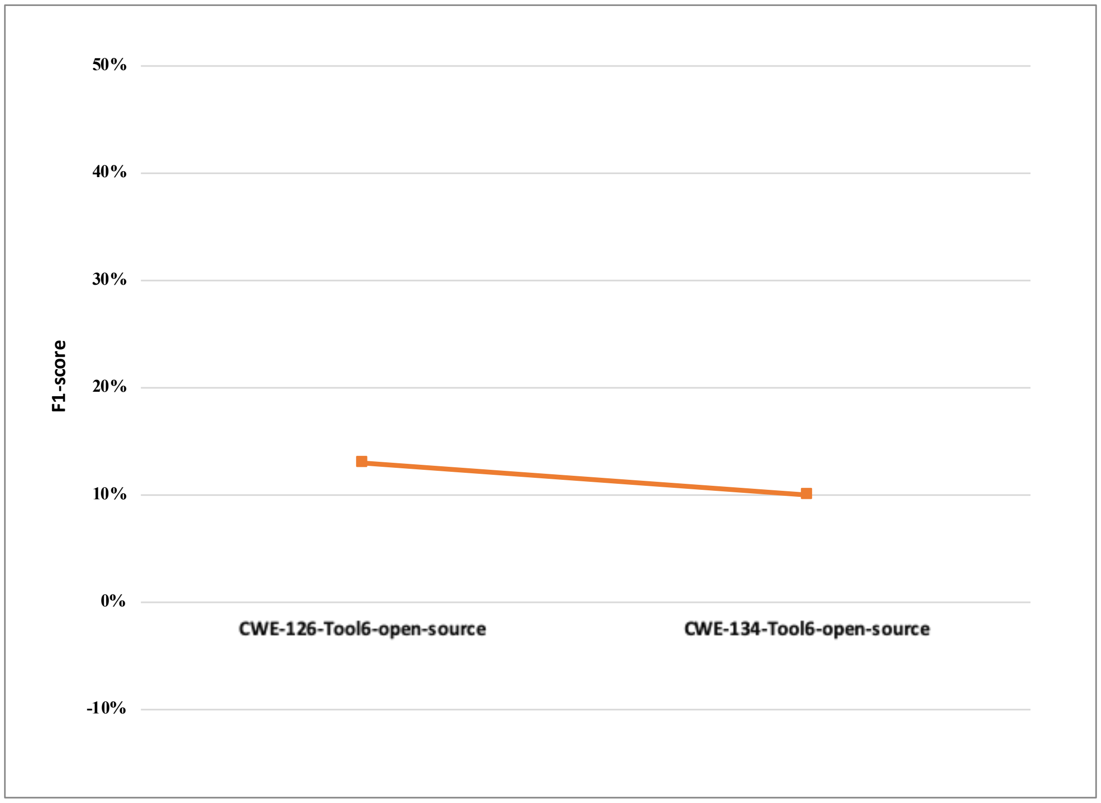


Fig. 5.11.: Result of a Baseline Model Trained
Using α and the Juliet Test Suite

In the next experiments, we did not consider α approach to build the RF classifier due to the low value of the F1-score. In the next experiment, we investigated the effect of β in weighing the Juliet test suite warnings.

5.5.3.5 **Experiment #5: Baseline Model Trained Using β and Juliet Suit**

As shown in Figure 5.12, we used β only to weight the source domain instances. Therefore, to compute the value of the β, we need to compute the value of $P_s(y_i^s|x_i^s)$ from the source domain. Likewise, we utilized the labeled target domain instances to compute the value of $P_t(y_i^s|x_i^s)$.

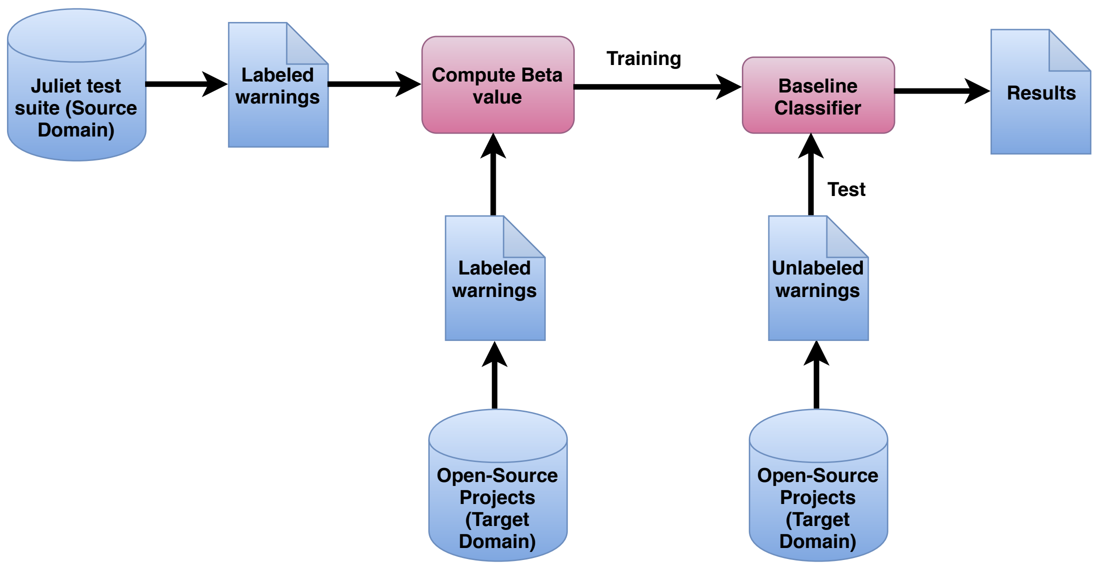


Fig. 5.12.: Using a Baseline Model Trained Using β and the Juliet Test Suite to Rank Open-Source Software Warnings

In another words, a logistic regression model was learned from the labeled target domain to compute $P_t(Y|X)$ as follows.

$$\theta_t = \underset{\theta_t}{\text{argmin}} \left[-\sum_{i=1}^{N_{t,l}} \ln P(y_i^{t,l}|x_i^{t,l}, \theta_t) + \lambda \left\|\theta_t\right\|^2 \right] \tag{5.20}$$

To compute $P(y|x, \theta_t)$, we used the following formula.

$$P(y|x, \theta_t) = \frac{exp(\theta_{t,y}^T \mathbf{x})}{\sum_{y' \in \mathcal{Y}} exp(\theta_{t,y'}^T \mathbf{x})} \tag{5.21}$$

Next, the trained model was used to estimate $P_t(y_i^s|x_i^s)$ as follows.

$$\begin{aligned} P_t(y_i^s|x_i^s) &\approx P(y_i^s|x_i^s, \theta_t) \\ &= \frac{exp(\theta_{t,y_i^s}^T \mathbf{x}_i^s)}{\sum_{y' \in \mathcal{Y}} exp(\theta_{t,y'}^T \mathbf{x}_i^s)} \end{aligned} \tag{5.22}$$

Finally, the value of the β_i can be computed as follows.

$$\begin{aligned} \beta'_i &= \frac{P(y_i^s|\mathbf{x}_i^s, \theta_t)}{P(y_i^s|\mathbf{x}_i^s, \theta_s)} \\ C &= \sum_{i=1}^{N_s} \beta'_i \\ \beta_i &= \frac{\beta'_i}{C} \end{aligned} \tag{5.23}$$

Lastly, we set $\lambda_s = 1$, $\lambda_{t,l} = 0$,and $\lambda_{t,u} = 0$. So, the final formula looks like below.

$$f_t^* = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{N_s} \beta_i L(x_i^s, y_i^s, f) \tag{5.24}$$

Furthermore, only the weighted source domain instances were used to train the RF classifier.

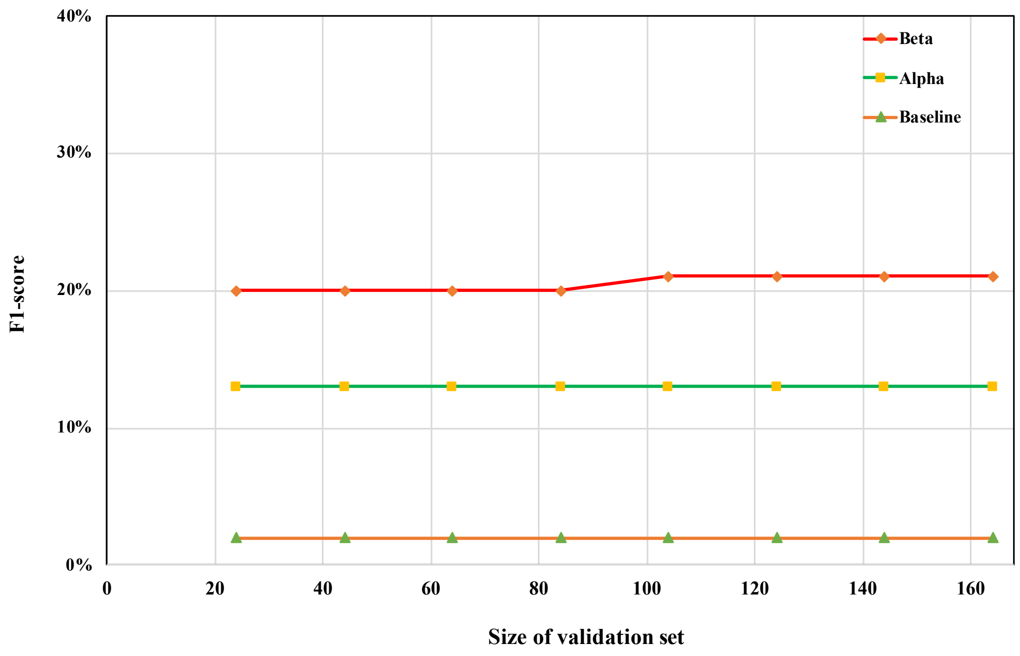


Fig. 5.13.: Evaluate the Performance of the Source Baseline Model, Baseline Model with β, and Baseline Model with α for CWE-126

From Figures 5.14 and 5.13, we can conclude that the β approach outperforms the baseline model and the α approach. On the other hand, increasing the number

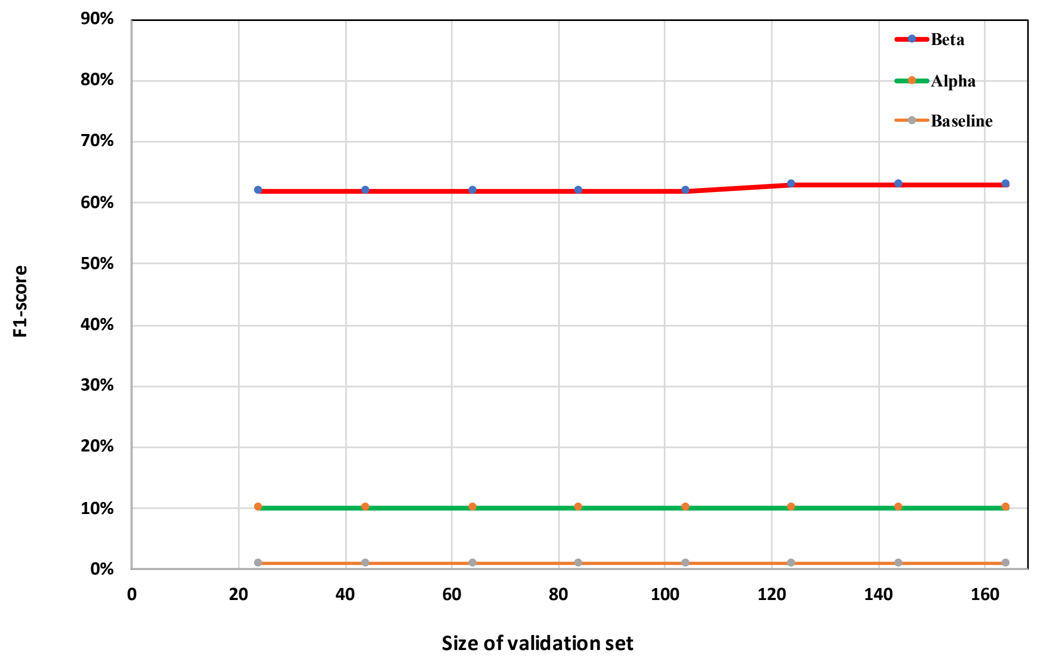


Fig. 5.14.: Evaluate the Performance of the Source Baseline Model, Baseline Model with β, and Baseline Model with α for CWE-134

of labeled warnings from the target domain can improve the performance of the β approach due to the need for the labeled target warnings to compute the value of the β. From these observations, we can infer that the difference between the Juliet test suite warnings and open-source software project warnings comes more from the difference in the conditional distribution. In the next experiment, we explored the effect of using both β and the labeled target domain to train the RF classifier and check whether this setting will improve the classifier performance.

5.5.3.6 **Experiment #6: Baseline Model Trained Over D_s and $D_{t,l}$ Using β**

In most of the previous experiments, the value of λ_s was set to 1 and the value of $\lambda_{t,l}$ was set to 0. In this experiment, the value of both λ_s and $\lambda_{t,l}$ was set to 1 and the value of $\lambda_{t,u}$ set to 0. In other worlds, we used the labeled warning from the open-source software projects to estimate β to weight the Juliet test suite warnings. Then, we combined the weighted warnings with the labeled open-source software project warnings to train RF classifier.

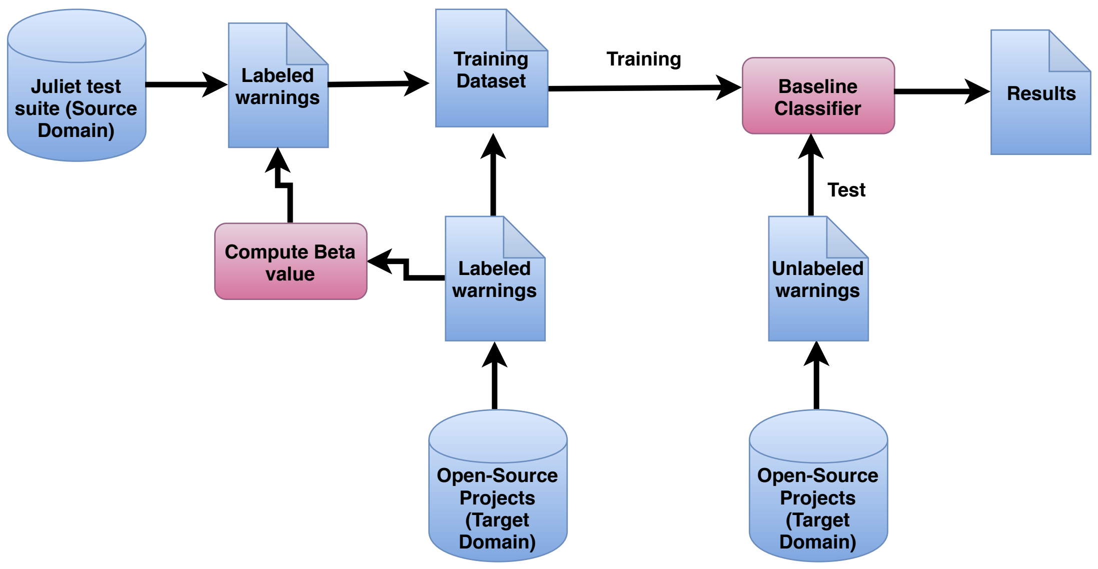


Fig. 5.15.: Using a Baseline Model Trained Using β, D_s, and $D_{t,l}$ to Rank Open-Source Software Warnings

The formula that we used in this experiment ia as follows [97].

$$f_t = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \left[\lambda_s \sum_{i=1}^{N_s} \beta_i L(x_i^s, y_i^s, f) + \lambda_{t,l} \sum_{i=1}^{N_{t,l}} L(x_i^{t,l}, y_i^{t,l}, f) \right] \tag{5.25}$$

From Figures 5.16 and 5.17, we can conclude that re-weighting the Juliet test suite warnings using β and utilizing the labeled target domain warnings to train the RF classifier will outperform the other proposed models even when the validation dataset is small (has only 24 warnings).

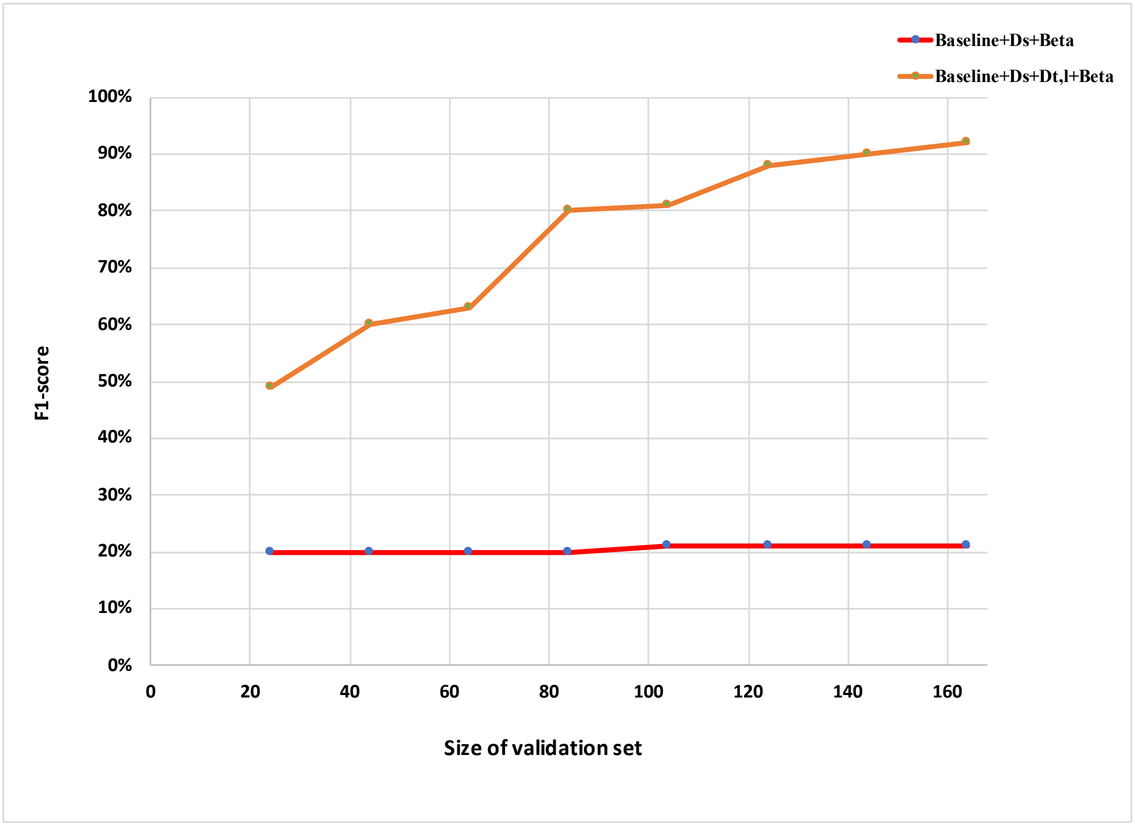


Fig. 5.16.: Evaluate the Performance of the Baseline Model with β Only and Baseline Model Trained Over D_s and $D_{t,l}$ using β for CWE-126

Based on the observations from our experiments, we can conclude that using the β to re-weight the Juliet test suite warnings and the labeled open-source software warnings to train an optimal RF classifier outperforms the other proposed setting, even though the difference between the target and source domains come from the conditional distribution. Therefore, we used this model as an optimal classifier to rank the false positive warnings aggregated from the real-world applications in the next sections.

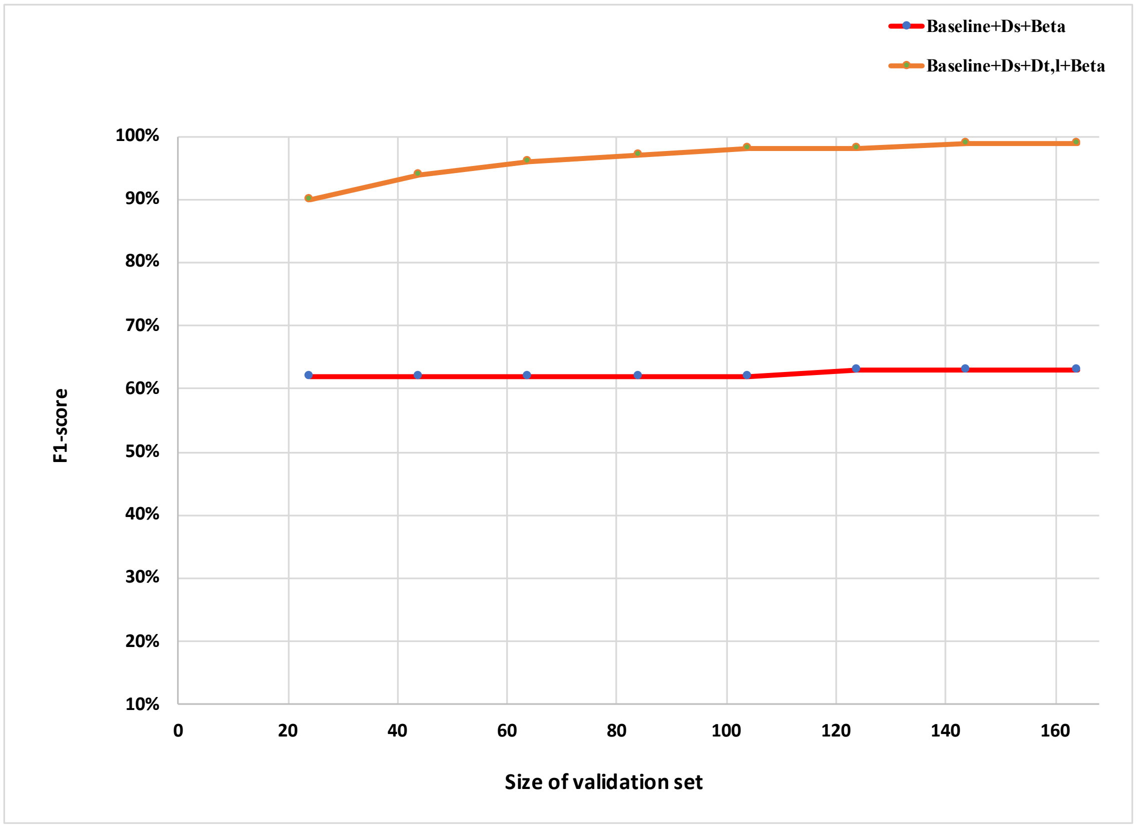


Fig. 5.17.: Evaluate the Performance of the Baseline Model with β Only and Baseline Model Trained Over D_s and $D_{t,l}$ using β for CWE-134

5.5.4 **Discussion**

This section discusses our experimental results for ranking SCA tool warnings in terms of different software engineering metrics.

5.5.5 **Most Important Software Engineering Metrics**

For our research, we are interested in building a framework that can rank the SCA tool warnings for a given function using a classifier model, but we also interested in finding which of the software engineering metrics are highly correlated with the true and false positives. Figure 5.18 and 5.19 show the relevant software engineering metrics that we identified in each open-source dataset. The most frequent selected software engineering metrics among the two datasets were: CountLineComment, CountLineCode, CountLineCodeExe, CountLineCodeDecl, CountInput, CountOutput, and Cyclomatic Strict.

Figure 5.18 showcases the most important software engineering metrics for the dataset called *CWE-126-Tool6-open-source*. From this figure we can observe that the most influenced software engineering metrics for classification and ranking the CWE-126 false warnings were the Complexity metrics (MaxNesting, Cyclomatic, and CountPath). Likewise, the some Coupling and Volume metrics (CountInput, CountLineInactive, CountOutput, and CountLineBlank) take a second place in impact on classification warnings.

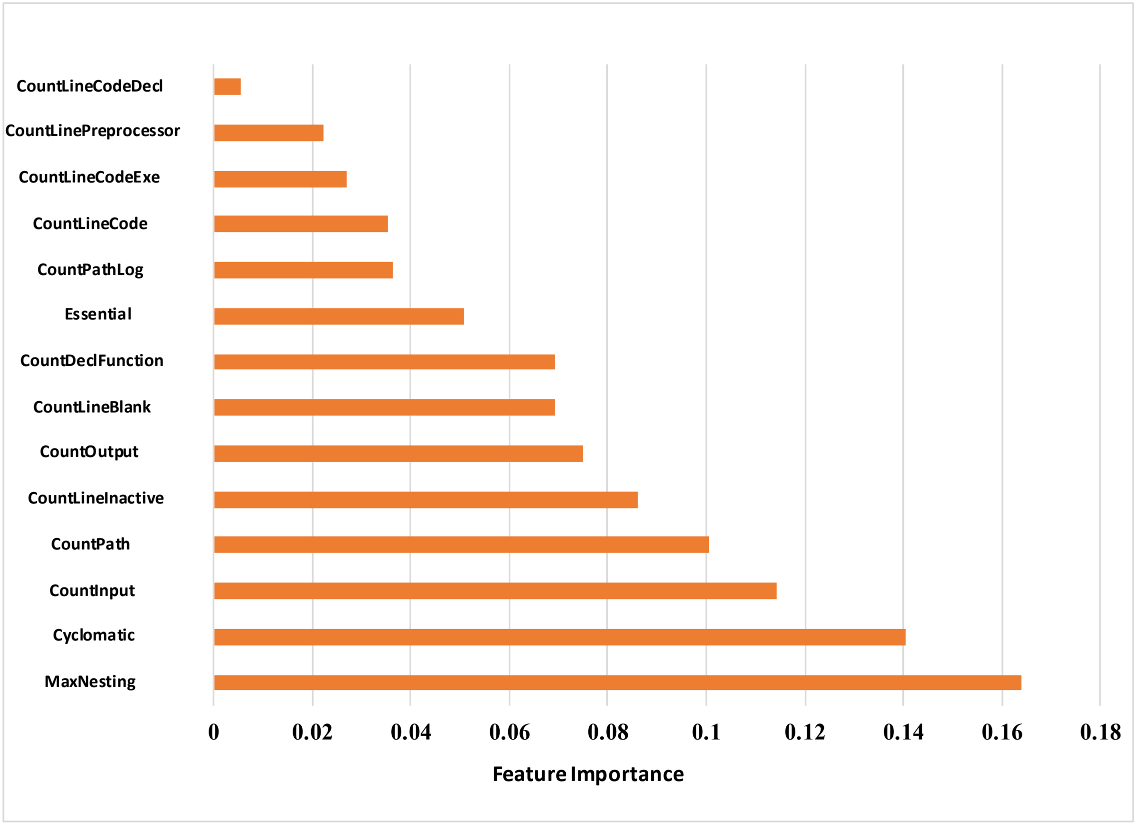


Fig. 5.18.: Important Software Engineering Metrics for CWE-126-Tool6-open-source

Figure 5.19 showcases the most important software engineering metrics for the dataset called *CWE-134-Tool6-open-source*. From this figure we can observe that the most influenced software engineering metrics for classification and ranking the CWE-134 false warnings were the Volume metrics (AltCountLineBlank, CountLineComment, CountLineBlank, and RatioCommentToCode). Likewise, the c]Complexity and Coupling metrics (Essential, MaxNesting, and CountInput) take a second place in impact on classification warnings.

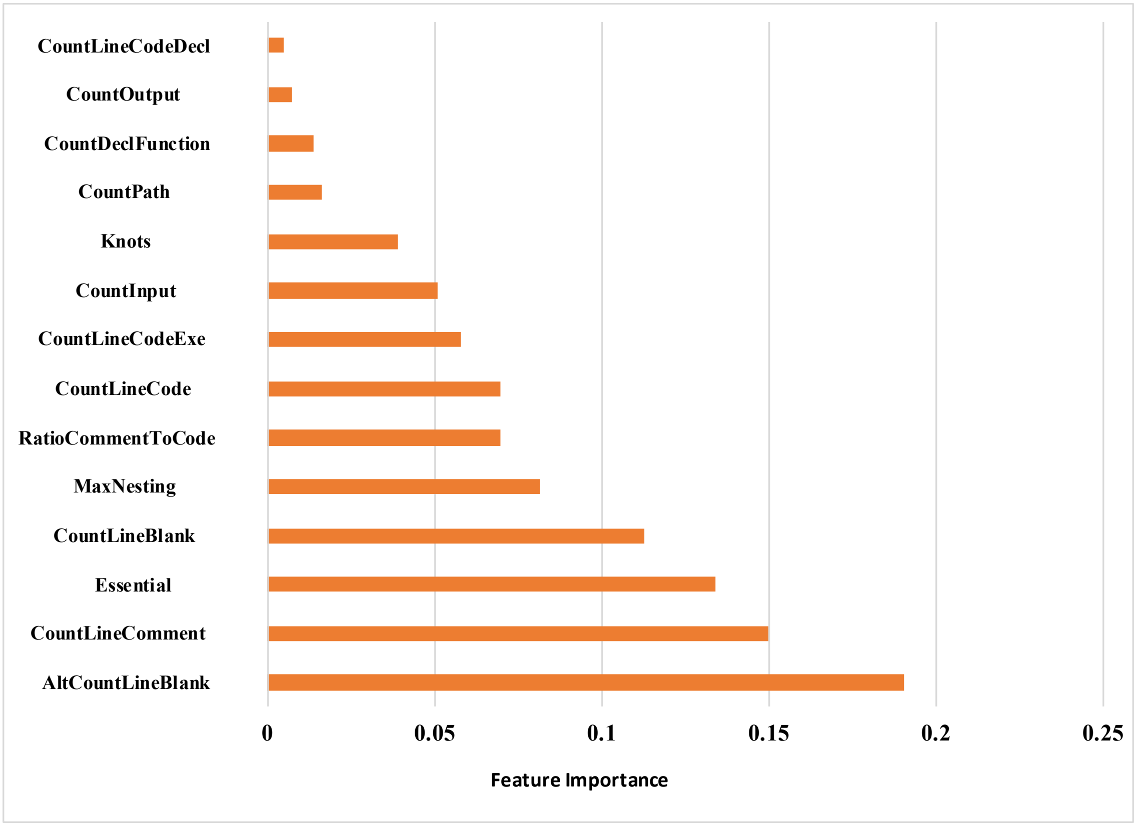


Fig. 5.19.: Important Software Engineering Metrics for CWE-134-Tool6-open-source.

From Figure 5.19 and Figure 5.18, we cannot determine what is the set of the most frequent selected software engineering metrics among the two datasets. In other words, there is no particular type of software engineering metrics that will be highly correlated with the Tool6 warnings across the two CWEs. It depends on what CWE is under investigation and the structure of the source code.

5.5.6 **Reorder Warnings**

To rank the emitted warnings from the open-source software, we use the probability computed by the optimal RF classifier. The probability value determines if the warning represents a potential defect in the given source code based on a specific type (*e.g.*, buffer over-read). The warnings that have probability value greater than threshold for a specific type of warnings such as CWE-126 is considered as true positives, while the warnings that have probability value less than or equal threshold is considered as false positives. This threshold was computed from the training datasets. Table 5.5 presents the threshold that we used in this work.

Table 5.5.: Threshold for each CWEs

Dataset Name	**Threshold**
CWE-126-Tool6-open-source	0.45
CWE-134-Tool6-open-source	0.36

To evaluate the proposed framework we follow the methodology presented by Heckman and Williams [29]. The output of the optimal ranking algorithm is a list that contains the true positive warnings at the top and all the false positive warnings at the bottom. In this work, generating a ranking list close to the optimal ranking list was are main objective. A secondary objective was to outperform the random ranking algorithm, which randomly shuffles the warnings generated from the open-source software.

To assess our work, we measured the strength of association between the optimal and the generated ranking list and the direction of the relationship using the Kendall and Spearman rank correlation measurement. A value of a positive one indicates a perfect degree of association between the optimal and the generated ranking list, which we tried to achieve.

Table 5.6 presents the Spearman rank correlation values between the optimal ranking algorithm and the proposed framework. We used the Cohen's standard to determine the strength of the relationship between the generated and optimal. From this table we can observe that all the correlation values were positive and greater than, or equal to 0.877. This means there is a strong match between the optimal ranking algorithm and the proposed framework. This also means that our framework outperforms the previous studies.

Table 5.6.: Spearman rank correlation comparing with optimal

Dataset Name	**Proposed Framework**
CWE-126-Tool6-open-source	0.970
CWE-134-Tool6-open-source	0.877

Table 5.7 presents the Spearman rank correlation values between the random ranking algorithm and the proposed framework. From this table we can observe that the correlation value between the proposed framework and the random ranking algorithm are negative for the dataset that represents CWE-126 warnings. This conclusion shows that there is no match between the random ranking algorithm and the proposed framework. In other worlds, there is no similar ordering of the false positive warnings between the algorithms.

Likewise, the correlation value between the random ranking algorithm and the proposed framework was positive but less than 0.10 for the dataset that represents CWE-134 warnings. This correlation value declares that there is a weak correlation between the random ranking algorithm and the our framework based on the Cohen's standard. Finally, we can infer that the our framework also outperforms the random ranking algorithm.

Table 5.7.: Spearman rank correlation comparing with random

Dataset Name	**Proposed Framework**
CWE-126-Tool6-open-source	-0.036
CWE-134-Tool6-open-source	0.0065

5.5.7 **Threat to Validity**

For this work, the threat to validity is related to the number of SCA tools used to evaluate the framework. For example, we did not include any of the state-of-the-art commercial tools because they are not freely available. We will work to address this threat in our future work (see Chapter 6).

5.6 **Summary of Contributions**

In this chapter, we have presented the Static Code Analysis Tool's Warnings Ranking (SCATWR), which is a framework to rank the warnings generated for real-world source code using software engineering metrics value. The following are the key contributions of the SCATWR.

- Proposing a framework to mitigate the false positive issue by prioritizing the SCA tool warnings for the purpose of guiding developers toward the most serious ones.
- Using software engineering metrics to rank the SCA tool warnings.
- Using instance weighting technique to transfer the RF classifier trained using Juliet test suite to rank the warnings collected from the open-source software.

6. CONCLUDING REMARKS

In this dissertation, we have proposed approach which can be used to improve the SCA tools performance using the ML techniques based on the source code metrics value. First, we presented the Static Code Analysis Tools Evaluator (SCATE) framework. SCATE evaluates the SCA tools in terms of source code metrics by running them either locally or remotely using the SWAMP. This framework uses the Understand static tool to compute the Volume, Complexity, and Object-Oriented metrics from the highlighted source code by the SCA tools. Likewise, the framework was used to convert the given source codes into a set of attribute files (ARFF).

Second, we presented the Static Code Analysis Tool Warnings Classification (SCATWC) framework, which is a framework to auto-construct a number of predictive models (classifiers) using four of machine learning techniques (KNN, SVM, RF, and RIPPER) from eight generated datasets. Each dataset represents a number of specific CWE test cases and a SCA tool's warnings. Where the dataset features show the degree of complexity and coupling for the given source code (synthetic source code) and the label or class value represents the type of SCA tool warning (true positive, false positive, and false negative) for this source code. This framework uses F1-score to evaluate the performance of each classifier. This metric shows that the Random Forest classifier outperforms the rest of the classifiers.

On the other hand, we used this framework to evaluate which of source code metrics are highly correlated with the true positive, false positive, and false negative warnings generated by an SCA tool. We then described SCATWR a proposed approach for ranking the SCA tool warnings collected from a number of open-source software projects using both the source code metrics and the Domain Adaptation (DA) technique (instance weighting technique). The following is a summary of lessons learned from the research work presented in this dissertation and some future research directions.

- Based on our current results for SCATE framework and the case study that we used choosing an SCA tool for a given source code depends on several important factors, such as the weakness the developers want to test for, the value of one or more of the source code metrics, and finally the structure of the given source code. For example, Tool1 does not perform well with the source code that has a high number of CountPath for CWE-476. Tool 1, however, performs better when the real-world source code (Xerces-C++) has a high number of paths.

- The experimental results of SCATE framework show that there is a relationship between the number of the uncovered flaws by SCA tools and the value of the software engineering metrics.

- Five of SCA tools that support both the C++ and Java programing language and also its rules mapping to CWEs have been evaluated using SCATE framework. In the future, we plan to extend this research to cover more open-source SCA tools that analyze Python, Ruby, C++, and Java source code.

- SCA tools results report a large number of false warnings (*e.g.*, false positive). The manual inspections of the false warnings is an unavoidable, time-consuming, and costly process.

- The overall results of the SCATWC show that the performance of the RF technique is the best on average across the other examined ML techniques. Its average F1-score is 90.4%, while the performance of the RIPPER technique is the second best one. Its average F1-score is 87.7%.

- The CountInput, Knots, CountOutput, CountPath, Cyclomatic, and Essential were considered as the most important software engineering metrics over the eight datasets to predict the behavior of SCA tools against a given synthetic source code.

- False positive and false negative warnings can be reduced if the developers rewrite their source code in a way that reduces source code complexity, coupling, and usage of global variables.
- The overall results of SCATWR framework show that the proposed framework determines a threshold value where the warning that has a confidence value larger than the threshold value can be considered as a true positive warning. On the other hand, the developers do not need to check the warning that has a confidence value less than the threshold value since the proposed framework considers this warning as false positive.
- RF technique was used to build two different ranking models for two common weaknesses (CWE-134 and CWE-126). Both of the models outperform the random ranking algorithm and generate a ranking list that has a high positive correlated with the optimal ranking algorithm.
- We only used a single SCA tools to validate SCATWR framework. But, we still need to investigate applying the framework across different SCA tools (including commercial tools), and against more CWEs. This will help us understand if the framework is both tool and CWE independent.

For future research efforts, we will apply our approach to source code from various open-source and commercial software projects. Likewise, we are planning to compute the software engineering metrics at the line level in order to reduce the number of contradictory data points in the generated datasets. Lastly, we plan to extend our approach by covering more SCA tools and using other advanced ML techniques.

REFERENCES

REFERENCES

[1] G. McGraw, "Software security," *IEEE Security Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[2] G. McGraw, *Software Security: Building Security In.* Addison-Wesley Professional, 2006.

[3] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3106739

[4] S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. K. Mandal, Eds., *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014 - Volume 1, Bhubaneswar, Odisa, India, 14-15 November 2014*, ser. Advances in Intelligent Systems and Computing, vol. 327. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-11933-5

[5] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Softw., Pract. Exper.*, vol. 24, pp. 197–218, 1994.

[6] 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.

[7] E. Alikhashashneh, R. Raje, and J. Hill, "Using software engineering metrics to evaluate the quality of static code analysis tools," in *Proceedings of the 1st International Conference on Data Intelligence and Security.* South Padre Island, TX, USA: IEEE Computer Society, 2018.

[8] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1052883.1052895

[9] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, July 2008.

[10] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," in *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 470–481.

[11] M. Kulenovic and D. Donko, "A survey of static code analysis methods for security vulnerabilities detection," *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1381–1386, 2014.

[12] "Findbugs - find bugs in Java programs," http://findbugs.sourceforge.net/, accessed: 2016-09-19.

[13] "Pylint star your python code!" https://www.pylint.org, accessed: 2019-0-23.

[14] "Klocwork-faster delivery of secure, reliable, and conformant code," https://www.roguewave.com/products-services/klocwork, accessed: 2019-05-19.

[15] J. Novak, A. Krajnc, and R. ï£¡ontar, "Taxonomy of static code analysis tools," in *MIPRO, 2010 Proceedings of the 33rd International Convention*, 2010, pp. 418–422.

[16] Wikipedia, "List of tools for static code analysis — Wikipedia, the free encyclopedia," 2016, [Online; accessed 13-September-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=739038439

[17] A. Ramos, "Evaluating the ability of static code analysis tools to detect injection vulnerabilities," Ph.D. dissertation, Umeå university, 2016.

[18] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, pp. 88–90, 2012.

[19] R. K. McLean, "Comparing static security analysis tools using open source software," in *Proceedings of IEEE 6th International Conference on Software Security and Reliability Companion*, 2012, pp. 68–74.

[20] J. Steinberg, *Apache OpenOffice. Org 3. 4: Using Math.* CreateSpace Independent Publishing Platform, 2013.

[21] K. Dooley and I. Brown, *Cisco IOS Cookbook, 2nd Edition Field-Tested Solutions to Cisco Router Problems.* O'Reilly Media, 2007.

[22] G. Lyon, *Nmap Network Scanning: Official Nmap Project Guide to Network Discovery and Security Scanning.* Insecure.Com, LLC, 2008.

[23] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit.* Syngress, 2006.

[24] 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*, ser. CISR '14'. New York, NY, USA: ACM, 2014, pp. 17–20.

[25] D. Coupal and P. Robillard, "Factor analysis of source code metrics," *Journal of Systems and Software*, vol. 12, no. 3, pp. 263–269, 1990.

[26] G. Botterweck and C. Werner, *Mastering Scale and Complexity in Software Reuse: 16th International.* Springer, 2017.

[27] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Static Analysis.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[28] C. Boogerd, "Prioritizing software inspection results using static profiling," in *Proc. of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06.* In International Workshop on Source Code Analysis and Manipulation, 2006.

[29] 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*, ser. ESEM '08. New York, NY, USA: ACM, 2008.

[30] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007.

[31] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, "A framework to compare alert ranking algorithms," in *in "19th Working Conference on Reverse Engineering*, 2012.

[32] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[33] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct 2016, pp. 157–166.

[34] OWASP.

[35] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting," *Journal of Software: Practice and Experience*, vol. 43, no. 3, pp. 259–279, 2012.

[36] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools (extended version)," Linköping University, Institute of technology, Link ÌĹoping, Sweden, Tech. Rep., 2008.

[37] V. Barstad, M. Goodwin, and T. Gjø sæter, "Predicting source code quality with static analysis and machine learning," *Norsk Informatikkonferanse*, 2014.

[38] U. Yuksel and H. Sözer, "Automated classification of static code analysis alerts: A case study." in *ICSM.* IEEE Computer Society, 2013, pp. 532–535.

[39] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools," in *Workshop on Machine Learning and Programming Languages*, 2017.

[40] Z. P. Reynolds, A. B. JayanthIndiana, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill, "Identifying and documenting false positive patterns generated by static code analysis tools," in *Proceedings of the 4th International Workshop on Software Engineering Research and Industrial Practice*, 2017.

[41] O. Tripp, S. Guarnieri, M. Pistoia, and Y. A. Aravkin, "ALETHEIA: improving the usability of static security analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[42] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.

[43] 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*, ser. ICSE '08. New York, NY, USA: ACM, 2008.

[44] L. Wei, Y. Liu, and S.-C. Cheung, "Oasis: prioritizing static analysis warnings for android apps based on app user reviews," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 08 2017, pp. 672–682.

[45] P. Black, I. T. L. N. I. of Standards, and Technology), *Juliet 1.3 Test Suite: Changes from 1.2*, ser. NIST technical note; NIST tech note; NIST TN. U.S. Department of Commerce, National Institute of Standards and Technology, 2018.

[46] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," in *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[47] A. Ribeiro, P. Meirelles, N. Lago, and F. Kon, "Ranking source code static analysis warnings for continuous monitoring of floss repositories," in *Open Source Systems: Enterprise Software and Solutions*. Cham: Springer International Publishing, 2018.

[48] H. K. Brar and P. J. Kaur, "Static analysis tools for security : A comparative evaluation," 2015.

[49] L. M. R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill, "Toward modeling the behavior of static code analysis tools," in *Proceedings of the 9th Cyber and Information Security Research Conference*, 2014.

[50] M. Livny and B. Miller, "The Case for an Open and Evolving Software Assurance Framework," http://continuousassurance.org/wp-content/uploads/2013/11/White-Paper-Evolving-Framework.pdf, 2016.

[51] U. D. of Homeland Security. (2018) The Common Weakness Enumeration (CWE) Initiative. http://cwe.mitre.org/.

[52] MITRE, "Cwe-416: Use after free," https://cwe.mitre.org/data/definitions/416.html, 2018, accessed: 2018-06-15.

[53] ——, "Cwe-124: Buffer underwrite ('buffer underflow')," https://cwe.mitre.org/data/definitions/124.html, 2018, accessed: 2018-06-15.

[54] ——, "Cwe-369: Divide by zero," https://cwe.mitre.org/data/definitions/369.html, 2018, accessed: 2018-06-15.

[55] Center for Assured Software, National Security Agency, "Juliet Test Suite v1.2 for C/C++ User Guide," 2011.

[56] ——, “Juliet Test Suite v1.1 for Java User Guide,” 2011.

[57] Sun Micro Systems, *Java Servlet Specification* , 3.0 ed., SUN, Dec. 2009.

[58] ——, *Java Server Pages Specification* , Version 2.1 ed., SUN, May 2006.

[59] K. Goseva-Popstojanova and A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities,” *Inf. Softw. Technol.*, vol. 68, no. C, pp. 18–33, Dec. 2015. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.08.002

[60] MITRE, “Cwe-457: Use of uninitialized variable,” https://cwe.mitre.org/data/definitions/457.html, 2018, accessed: 2018-06-15.

[61] “Cwe-476: Null pointer dereference,” https://cwe.mitre.org/data/definitions/476.html, accessed: 2017-09-15.

[62] Scitools, “Understand tool,” https://scitools.com/, 2016, accessed: 2018-03-15.

[63] C. Artho and A. Biere, “Applying static analysis to large-scale, multi-threaded java programs.” in *In Proceedings of the 13th Australian Conference on Software Engineering*, 2001.

[64] A. Ali, S. M. H. Shamsuddin, and A. L. Ralescu, “Classification with class imbalance problem: a review,” in *SOCO 2015*, 2015.

[65] V. Ganganwar, “An overview of classification algorithms for imbalanced datasets,” *International Journal of Emerging Technology and Advanced Engineering*, 2012.

[66] M. Hernández and S. Stolfo, “Real-world data is dirty: Data cleansing and the merge/purge problem,” *Data Min. Knowl. Discov.*, 1998.

[67] I. Steinwart and A. Christmann, *Support Vector Machines.* Springer Publishing Company, Incorporated, 2008.

[68] B. Dasarathy, *Nearest Neighbor (NN) Norms: Nn Pattern Classification Techniques.* IEEE Computer Society Press, 1991.

[69] L. Breiman, “Random forests,” *Mach. Learn.*, no. 1, pp. 5–32, 2001.

[70] A. Rajput, R. P. Aharwal, M. Dubey, S. Saxena, and M. Raghuvanshi, “J48 and jrip rules for e-governance data,” *International Journal of Computer Science and Security (IJCSS)*, 2011.

[71] M. Hall, “Correlation-based feature selection for machine learning,” Ph.D. dissertation, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1999.

[72] M. A. Hall, “Correlation-based feature selection for discrete and numeric class machine learning,” in *Proceedings of the Seventeenth International Conference on Machine Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[73] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient.* Springer Berlin Heidelberg, 2009, pp. 1–4.

[74] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, 2002.

[75] A. Fernández, V. López, M. Galar, M. del Jesus, and F. Herrera, "Analysing the classification of imbalanced data-sets with multiple classes: Binarization techniques and ad-hoc approaches," *Knowledge-Based Systems*, 2013.

[76] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society*, 1974.

[77] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, 2008.

[78] Y. Awad, M. Yashwant, A. Charles, and R. Indrajit, "To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy.* New York, NY, USA: ACM, 2016.

[79] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, 2008.

[80] MITRE, "Cwe-762: Mismatched memory management routines," https://cwe.mitre.org/data/definitions/762.html, 2018, accessed: 2018-06-15.

[81] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, "Handling imbalanced datasets: A review," *GESTS International Transactions on Computer Science and Engineering*, 2006.

[82] M. Bekkar, H. K. Djemaa, and T. A. Alitouche, "Evaluation measures for models assessment over imbalanced data sets," *Journal of Information Engineering and Applications*, 2013.

[83] E. Rich and K. Knight, *Artificial intelligence (2. ed.).* McGraw-Hill, 1991.

[84] G. Anderson, "Random relational rules," Ph.D. dissertation, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 2009.

[85] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[86] "Cwe-126: Buffer over-read," https://cwe.mitre.org/data/definitions/126.html, accessed: 2019-01-15.

[87] "Cwe-134: Use of externally-controlled format string," https://cwe.mitre.org/data/definitions/134.html, accessed: 2019-01-15.

[88] "Format string attack," https://www.owasp.org/index.php/Format_string_attack, accessed: 2019-06-05.

[89] "sklearn.metrics.f1_score," https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html, accessed: 2019-01-15.

[90] I. Witten, E. Frank, M. Hall, and C. Pal, *DATA MINING: Practical Machine Learning Tools and Techniques.* Todd Green, 2017.

[91] W. M. Kouw, "An introduction to domain adaptation and transfer learning," *CoRR*, 2018.

[92] M. Sugiyama, S. Nakajima, H. Kashima, P. v. Bünau, and M. Kawanabe, "Direct importance estimation with model selection and its application to covariate shift adaptation," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, ser. NIPS'07. USA: Curran Associates Inc., 2007, pp. 1433–1440. [Online]. Available: http://dl.acm.org/citation.cfm?id=2981562.2981742

[93] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *Journal of Statistical Planning and Inference*, vol. 90, no. 2, pp. 227 – 244, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378375800001154

[94] B. Zadrozny, "Learning and evaluating classifiers under sample selection bias," *Proceedings, Twenty-First International Conference on Machine Learning, ICML 2004*, vol. 2004, 09 2004.

[95] J. J. Jiang, "A literature survey on domain adaptation of statistical classifiers," 2007.

[96] J. Blitzer, "Domain adaptation of natural language processing systems," Ph.D. dissertation, Philadelphia, PA, USA, 2008.

[97] J. Jiang and C. Zhai, "Instance weighting for domain adaptation in NLP," in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 264–271. [Online]. Available: https://www.aclweb.org/anthology/P07-1034

[98] S. J. Pan, X. Ni, J.-T. Sun, Q. Yang, and Z. Chen, "Cross-domain sentiment classification via spectral feature alignment," in *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: ACM, 2010.

VITA

VITA

Enas Ahmad Alikhashashneh received her bachelor's degree from Al-Balqa' Applied University, Jordan in 2006 and her Masters' degree from Purdue University Indianapolis in 2017. Before joining Purdue University she worked as a Full-time lecturer at Yarmouk University. Upon receiving his Ph.D. from Purdue University Indianapolis she will join Yarmouk University as an assistant professor.