Ranking Source Code Static Analysis Warnings for Continuous Monitoring of Free/Libre/Open Source Software Repositories

Athos Coimbra Ribeiro

Thesis presented to the Institute of Mathematics and Statistics of the University of São Paulo in partial fulfillment of the requirements for the degree of Master of Science

Program: Computer Science
Advisor: Prof. Fabio Kon

During this work, the author was supported by CAPES

São Paulo
June 22nd, 2018
Ranking Source Code Static Analysis Warnings for Continuous Monitoring of Free/Libre/Open Source Software Repositories

Athos Coimbra Ribeiro

This version of the thesis includes the corrections and modifications suggested by the Examining Committee during the defense of the original version of the work, which took place on June 22nd, 2018.

A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Examine Committee:

- Prof. Dr. Marcelo Finger - IME-USP
- Prof. Dr. Gregorio Robles - UJRC
- Prof. Dr. Eduardo Guerra - INPE
I authorize reproduction and disclosure of this work through any conventional or electronic means for study and research purposes, provided that the source is mentioned.
Para Gabi.
Agradecimientos

I would like to thank my advisor, professor Fabio Kon, whose support was essential for all my research activities. It had been an honor to learn how research should be performed in computer science from Fabio. Professor Paulo Roberto Miranda Meirelles for his support and for the trust and confidence he has placed on me through the years. Dr. Paul E. Black and the Software Assurance Metrics and Tool Evaluation (SAMATE) team for all their patience and teachings during my stay at the US National Institute of Standards and Technology, which inspired my work. The Fedora Project community, for receiving our ideas in the Google Summer of Code project and giving us feedback on our work. We would also like to thank Nelson Lago for all his ideas and wonderful reviews.

I was funded by CAPES during this research extent.

David Silva became a core contributor to kiskadee code base through the Google Summer of Code program under the Fedora Project organization mentored by Athos Ribeiro.

The following undergraduate students of University of Brasília contributed to kiskadee’s source code and developed most of the current system front end, as of the date this document was written. They did so under a maintenance course taught by Prof. Paulo Roberto Miranda Meirelles. They are Adailson Santos, Daniel Moura, Danilo Barros, David Carlos, Eduardo Gomes, Eduardo Nunes, Eduardo Quintino, Fabio Teixeira, Gabriel Climaco, Gesiel Freitas, Lucas Andrade, Matheus Godinho, Matheus Mello, Maxwell Oliveira, Omar Junior, Rafael Rabetti, Thiago Moreira, Vitor Bertulucci, and Vitor Borges. Thank you for your valuable contributions.

Finally, I would like to thank my parents, Edivaldo and Rita, for always understanding and supporting me unconditionally.
Resumo

Athos Coimbra Ribeiro. **Ranking Source Code Static Analysis Warnings for Continuous Monitoring of Free/Libre/Open Source Software Repositories.** Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Embora exista grande variedade de analisadores estáticos de código-fonte disponíveis no mercado, tanto com licenças proprietárias, quanto com licenças livres, cada uma dessas ferramentas mostra melhor desempenho em um pequeno conjunto de problemas distinto, dificultando a escolha de uma única ferramenta de análise estática para analisar um programa. A combinação das análises de diferentes ferramentas pode reduzir o número de falsos negativos, mas gera um aumento no número de falsos positivos (que já é alto para muitas dessas ferramentas). Uma solução interessante é filtrar esses resultados para identificar os problemas com menores probabilidades de serem falsos positivos. Este trabalho apresenta **kiskadee**, um sistema para promover o uso da análise estática de código fonte durante o ciclo de desenvolvimento de software provendo relatórios de análise estática ranqueados. Primeiramente, **kiskadee** roda diversos analisadores estáticos no código-fonte. Em seguida, utilizando um modelo de classificação, os potenciais bugs detectados pelos analisadores estáticos são ranqueados conforme sua importância, onde defeitos críticos são colocados no topo de uma lista, e potenciais falsos positivos, ao fim da mesma lista.

Para treinar o modelo de classificação do **kiskadee**, realizamos uma pós-análise nos relatórios gerados por três analisadores estáticos ao analisarem casos de teste sintéticos disponibilizados pelo **National Institute of Standards and Technology** (NIST) dos Estados Unidos. Para tornar a técnica apresentada o mais genérica possível, limitamos nossos dados às informações contidas nos relatórios de análise estática das três ferramentas, não utilizando outras informações, como históricos de mudança ou métricas extraídas do código-fonte dos programas inspecionados. As características extraídas desses relatórios foram utilizadas para treinar um conjunto de árvores de decisão utilizando o algoritmo AdaBoost para gerar um classificador mais forte, atingindo uma acurácia de classificação de 0,8 (a taxa de falsos positivos das ferramentas utilizadas foi de 0,61, quando combinadas). Finalmente, utilizamos esse classificador para ranquear os alarmes dos analisadores estáticos nos baseando na probabilidade de um dado alarme ser de fato um bug no código-fonte. Resultados experimentais mostram que, em média, quando inspecionando alarmes ranqueados pelo **kiskadee**, encontram-se 5,2 vezes menos falsos positivos antes de se encontrar cada bug quando a mesma inspeção é realizada para uma lista ordenada de forma aleatória.

**Palavras-chave:** Análise estática de código-fonte. Qualidade de software. Engenharia de software.
Abstract

Athos Coimbra Ribeiro. Ranking Source Code Static Analysis Warnings for Continuous Monitoring of Free/Libre/Open Source Software Repositories. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2018.

While there is a wide variety of both open source and proprietary source code static analyzers available in the market, each of them usually performs better in a small set of problems, making it hard to choose one single tool to rely on when examining a program. Combining the analysis of different tools may reduce the number of false negatives, but yields a corresponding increase in the number of false positives (which is already high for many tools). An interesting solution, then, is to filter these results to identify the issues least likely to be false positives. This work presents kiskadee, a system to support the usage of static analysis during software development by providing carefully ranked static analysis reports. First, it runs multiple static analyzers on the source code. Then, using a classification model, the potential bugs detected by the static analyzers are ranked based on their importance, with critical flaws ranked first, and potential false positives ranked last. To train kiskadee’s classification model, we post-analyze the reports generated by three tools on synthetic test cases provided by the US National Institute of Standards and Technology. To make our technique as general as possible, we limit our data to the reports themselves, excluding other information such as change histories or code metrics. The features extracted from these reports are used to train a set of decision trees using AdaBoost to create a stronger classifier, achieving 0.8 classification accuracy (the combined false positive rate from the used tools was 0.61). Finally, we use this classifier to rank static analyzer alarms based on the probability of a given alarm being an actual bug. Our experimental results show that, on average, when inspecting warnings ranked by kiskadee, one hits 5.2 times less false positives before each bug than when using a randomly sorted warning list.

Keywords: Source code static analysis. Software quality. Software engineering.
List of Abbreviations

API  Application Programming Interface
CWE  Common Weakness Enumeration
CVE  Common Vulnerabilities and Exposures
FLOSS Free/Libre and Open Source Software
FN   False Negative
FP   False Positive
NIST National Institute of Standards and Technology
ROC  Receiver Operating Characteristic
SAMATE Software Assurance Metrics And Tool Evaluation
SATE Static Analysis Tool Exposition
SIG  Fedora Special Interest Group
TN   True Negative
TP   True Positive
UI   User Interface
UnB  University of Brasilia
List of Figures

2.1 Paper selection ................................................................. 5

3.1 FLOSS bug report workflow. .............................................. 11

3.2 kiskadee design overview. ................................................. 12

3.3 kiskadee UI: Package search page ........................................ 17

3.4 Kiskadee UI: List and versions of analyzed packages ............ 18

3.5 Kiskadee UI: Comprehensive list of analyzed packages .......... 18

4.1 Juliet test suite universe (J) and its subsets detected by each static analyzer $T_x$. .............................................. 22

4.2 Venn diagram for $\{J - (T_1 \cup T_2 \cup \ldots T_n)\}$ (grey area), where $J$ is the Juliet test suite universe and $T_x$ is the coverage of the static analyzer $x$. ................. 23

4.3 Venn diagram for $\{T_1 \cup T_2 \cup \ldots T_n\}$ (grey area), where $J$ is the Juliet test suite universe and $T_x$ is the coverage of the static analyzer $x$. ................. 23

4.4 Venn diagram for $\{T_1 \cap T_2 \cap \ldots T_n\}$ (grey area), where $J$ is the Juliet test suite universe and $T_x$ is the coverage of the static analyzer $x$. ................. 24

4.5 Venn diagram for $\{(T_1 \cap T_2) - T_3\}$ etc. (grey area), where $J$ is the Juliet test suite universe and $T_x$ is the coverage of the static analyzer $x$. ................. 24

5.1 Precision of each individual tool and aggregated tools (precision of all tools combined), compared with our predictive model results ..................... 33

5.2 AdaBoost classifier features importance: sum of the feature importances of each individual decision tree, divided by the total number of decision trees . 34

5.3 Number of actual flaws hit upon top-down inspection of a ranked list for different ranking approaches. kiskadee refers to the predictive model proposed in this study ......................................................... 35

5.4 Area under ROC curve versus the number of trees used for AdaBoost ... 37
List of Tables

3.1 Selected static analysis tools .................................................. 15
4.1 Number of Juliet test cases ....................................................... 25
4.2 Total number of warnings generated per tool .............................. 25
4.3 Labeled warnings per tool ....................................................... 26
4.4 Possible Assigned Warning Categories ..................................... 27
5.1 Mean training classification accuracy (cross-validation) ............... 31
5.2 Confusion matrix ................................................................. 32
5.3 Average number of cumulative false positives hit before each true positive . 35
Contents

1 Introduction .................................................. 1
   1.1 Objectives .............................................. 2
   1.2 Contributions .......................................... 3
   1.3 Structure of This Work ................................. 4

2 Structured Literature Review ............................... 5
   2.1 Related work ............................................ 6
      2.1.1 Existing data and terminology .................. 7
      2.1.2 Handling Static Analysis Warnings .............. 7

3 Continuous Static Analysis Tool Design ................. 11
   3.1 Features and System Overview ....................... 12
   3.2 Monitoring Software Repositories .................. 13
   3.3 Static Analyzers ...................................... 14
   3.4 Common Reporting Language ......................... 15
   3.5 API and Web User Interface ......................... 17
   3.6 Architecture and Implementation Details ........... 17
   3.7 Related Projects ..................................... 19
      3.7.1 Debile ........................................... 19
      3.7.2 mock-with-analysis ............................. 20
      3.7.3 Coala ............................................ 20

4 A Static Analysis Warnings Ranking Model .............. 21
   4.1 Data Extraction ....................................... 21
      4.1.1 Choosing Data Sources .......................... 21
      4.1.2 Collecting Labeled Warnings From Multiple Static Analyzers .... 25
      4.1.3 Extracting Features from Labeled Warnings ....... 26
   4.2 Arbitrating on Warnings Positiveness ................ 27
      4.2.1 Training Decision Trees with AdaBoost ........ 27
      4.2.2 Ranking Static Analysis Warnings ................ 29
Chapter 1

Introduction

Any reasonably large program in production has bugs, and minimizing their number and impact is an important aspect of software development. However, finding bugs is often a difficult task, especially when they do not manifest themselves directly but result in security flaws.

Automated static analysis tools, also referred to as static analyzers, can alleviate this problem by examining the source code of computer programs for pre-determined properties, particularly security flaws. In theory, source code static analysis can explore abstractions of all possible program behaviors, which is not feasible with software testing (Hovemeyer and Pugh, 2004). Hence, this class of tools has the potential to find problems that tend to manifest themselves infrequently and are, therefore, harder to detect through other methods (Paul E Black, 2009), which makes their usage a complementary approach to automated testing. Once aware of a possible anomaly, developers can inspect the suggested section of the program and take action when necessary, patching the flawed code even before it reaches production releases.

Such tools show great promise but, due to the undecidable nature of the problem (Landi, 1992), static analyzers commonly use heuristics or approximations to determine properties for a given code construct, which frequently induces them to make inaccurate assumptions regarding the behavior of the program under evaluation. These assumptions may either lead the tool to generate a warning¹ that does not reveal a real flaw in the analyzed program, i.e., a false positive, or to fail generating a warning to a real flaw, i.e., a false negative. Since static analysis enumerates many execution paths, static analyzer reports frequently contain an excessive amount of information, which often includes a substantial amount of false positives due to the undecidability problem.

Empirical studies show that even carefully designed static analyzers may generate warnings at false positive rates that can exceed 30% (Kremenek, Ashcraft, et al., 2004). Such false alarms require manual inspection, increasing the effort of analyzing tool reports when they include significant amounts of false positives (T. B. Muske et al., 2013; S. S. Heckman, 2007), hindering the usage and general adoption of automated source code analyzers (Johnson et al., 2013), which can lead static analyzers to be discarded as irrelevant (Kremenek and Engler, 2003). A substantial improvement in this area could foster a more extensive inclusion of static analysis in software development cycles which, in turn, could help reduce the number of bugs and security flaws in many different kinds of

¹When a static analyzer finds an anomaly, it produces an alarm or warning. In this paper, we use both terms interchangeably to refer to a location in the source code pointed as a problematic site by a static analyzer. We use the term report to refer to a set of warnings generated in a single run of a static analyzer for a given target program.
The mere reduction of the false positive rates obtained from static analysis — and, consequently, of the amount of counterproductive information generated by static analyzers — can smooth the process of manually verifying analyzer reports. This idea has prompted the elaboration of several studies that propose either pruning or ranking the warnings from static analysis reports to deal with false alarms (Ruthruff et al., 2008; Kremenek, Ashcraft, et al., 2004; S. Heckman and Williams, 2009; Jung et al., 2005). These studies usually combine information extracted from both the static analysis reports as well as the project being examined to derive conclusions and infer if an alarm is an actual software flaw. In contrast, we hypothesize that the incorporation of various specialist tools, i.e., the use of multiple static analysis tools in conjunction, makes it possible to produce a reasonably accurate predictive model without the need for any knowledge about the examined source code. This is advantageous because this model could serve as an entry point for enhanced static analysis reports without the need of meticulous pre-processing and training steps for each inspected program. Seeing that some tools perform better than others in specific tasks due to different analysis methods, the usage of multiple static analysis tools might also increase static analysis coverage, reducing the number of false negatives that could occur by using a single tool (Paul E Black, 2009). Naturally, using multiple static analyzers to inspect the same program source code also inflates the absolute number of false positives. This downside, however, is an acceptable compromise given the corresponding improvement in the rate of false positives — the core of this work.

Furthermore, some static analysis techniques require significant processing to perform the analysis (Pan et al., 2006), which makes the whole static analysis process expensive. The latter issue also impacts research on static analysis, since much processing is needed when one wants to analyze how different tools perform on a large software corpus, study false alarm rates, or investigate vulnerabilities behavior. Johnson et al., 2013 conducted a survey to understand static analysis adoption on development processes. The authors found that false positives and the way warnings are presented to the user are some of the reasons why static analyzers are underused, and that improving the integration of these tools to the development process with a more intuitive presentation of flaws and better explanations could help increase their adoption. Automating the static analysis process would also enable the creation of static analysis report databases that could be queried by developers before new software releases and, if made public, would also provide more information to researchers, enabling more studies, or allowing them to focus on the actual research rather than waiting for static analyzers to complete their analyses.

1.1 Objectives

In this study, we present kiskadee, a system designed to support continuous static analysis in software repositories using multiple static analyzers to generate, store, and rank warnings. Since each static analyzer has its unique report format, kiskadee converts all reports generated to a common report language. By running multiple static analyzers on the same code base, kiskadee reduces the number of false negatives in the analysis. To address false positives, kiskadee ranks warnings in the static analysis reports using the AdaBoost algorithm’s (Freund et al., 1999) classification probabilities. Warnings with the highest rank are more likely to indicate real and more critical software flaws and warnings with the lowest rank are more likely to be false positives. In this context, a warning is a single issue produced by a static analyzer. Finally, kiskadee stores the ranked static analysis reports in a database. The ranked reports in the database are made available to kiskadee users,
1.2 Contributions

This work provides the following three major contributions:

**C1 Continuous static analysis tool.** *Kiskadee* is a tool able to continuously run different static analyzers in different software versions by monitoring software repositories through an extensible interface. It then stores the analysis reports into a database, using a common reporting format. If a kiskadee’s production instance is made publicly available, its database may aid both open source software developers and researchers. Developers will be able to query warnings reported for their projects and act on possible flaws in their source code base. Researchers will be able to use the database as a static analysis benchmark.

**C2 Method to rank warnings from different static analyzers based on criticality and veracity of the reports.** By answering our research questions, it will be possible to propose such method and rank the warnings in the *kiskadee* database, providing more value to the output of static analyzers.
1.3 Structure of This Work

The rest of this thesis begins with a thorough description of the literature review performed to situate our work with reference to the state of the art regarding source code static analysis usage and warnings processing (Chapter 2). To get readers familiar with source code static analysis terminology, Section 2.1.1 defines or explains essential concepts. We then have a closer view on the limitations in the quality of current static analysis tools reports and on how previous works make good use of source code analysis to minimize them (Section 2.1.2). As we will see, this entails significant effort for every single program to be analyzed, highlighting the utility of avoiding this step. In Chapter 3, we introduce the continuous static analysis tool developed during this work, provide details on its design, and discuss the reasons behind our implementation decisions. This last chapter presents our approach used to reduce false negative occurrences on static analysis reports and to provide a database of static analysis reports for different software versions, which is done by continuously running multiple static analyzers on software repositories. Now, it remains to describe our experience with the approach to mitigate false positive occurrences on static analysis reports, which is left for Chapter 4. In Section 4.1, we talk about how we normalized the data obtained from the static analysis tools for further processing and how we chose the features used to train our ranking model. In Section 4.2, we present our approach to process this data and build this model. We finally proceed to describe and discuss the experimental validation and obtained results in Chapter 5.
Chapter 2

Structured Literature Review

A structured literature review was conducted to identify how past works approach static analysis with focus on analysis of multiple versions of a given software, study of false alarms, and alarms classification. Figure 2.1 illustrates the design and application of this structured literature review. The IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)\(^1\) is known to be one of the most influent conferences for static analysis research and was selected for this review. SCAM provided us with a detailed survey on static analysis alarms processing, as discussed below, which gave us the necessary background to perform our research without the need to extend our literature review to additional conferences.

![Figure 2.1: Paper selection](image)

Starting with all SCAM papers published from 2001 (first edition) to 2017, three search expressions were defined for a first paper selection phase. Since we are interested in literature related to source code static analysis and bug classification to mitigate false positive occurrences, papers containing either static analysis, false positive or bug classification were labeled as matching papers (MP).

Then, we defined three categories of interest for a second paper selection phase. We defined the categories based on the topics related to our research goal, which, as described

\(^1\)ieee-scarm.org
in Chapter 1, involves running multiple static analyzers in different revisions of software to prioritize bugs and reduce false alarms. The categories defined were:

- Bug classification
- Use of multiple static analyzers
- False alarms in static analysis

If a bibliographic reference in any of the papers labeled as matching paper had a title suggesting that the paper could fit into any of the categories above, it was labeled as a matching paper bibliographic reference (MPBR). This inclusion of references was done to select outstanding work in the field, after noticing common bibliographic references in a significant number of papers labeled as matching papers. The abstract and conclusion sections in the papers labeled as either matching papers or matching paper bibliographic references were read, and the papers not related to any of the defined categories were discarded.

Finally, all the 36 papers not discarded were selected for this review, as shown in Figure 2.1. In special, T. Muske and Serebrenik, 2016 provided an overview of different techniques on how to handle static analysis alarms by successfully answering their research question: what are possible approaches for handling the static analysis alarms? In this survey, the authors classified part of the techniques as the automatic post-processing of the alarms, which includes ranking or classification of alarms. Given that this survey is strongly related to our study, we assessed the majority of the survey references under the related classifications, such as alarms ranking, to better position our work. Furthermore, other papers, like the ones recommended by the US National Institute of Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE) project  were reviewed as well (we did not include them in the structured review numbers). Appendix A presents a comprehensive list of the papers reviewed through the methodology here presented.

2.1 Related work

Evans and Larochelle, 2002 state that although security vulnerabilities are well understood, it is not a common practice to include techniques to detect or avoid them in the development processes and suggests that instead of solely relying on programmers, tools to prevent and detect software vulnerabilities should be built and integrated into software development cycles. It is also known that static analysis helps programmers identify software vulnerabilities before they reach production (Evans and Larochelle, 2002) in a cheaper way than solely conducting manual inspections (Johnson et al., 2013), thus our interest in static analyzers.

Several studies propose ways to reduce false positives, which are usually based on historical analysis (Penta et al., 2008; Spacco et al., 2006; Kim and Ernst, 2007) or some level of statistical analysis (T. B. Muske et al., 2013; Boogerd and Moonen, 2006; Kremenek and Engler, 2003; Ruthruff et al., 2008). The remainder of this section discusses related work among the reviewed papers selected during the structured literature review. First, we define important terminology that we will use throughout this research. Then, we discuss works related to static analysis, reduction of false positives in static analysis reports, and static analysis warnings ranking or classification.

\[ \text{samate.nist.gov/index.php/Bibliography.html} \]
2.1 | RELATED WORK

2.1.1 Existing data and terminology

The United States Committee on National Security Systems\(^3\) (CNSS) defines **software assurance** as the *level of confidence that software is free from vulnerabilities [...] and that the software functions in the intended manner* (INSTRUCTION, 2003). The National Aeronautics and Space Administration (NASA) defines software assurance as *the planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures*, which includes the disciplines of software safety and software verification and validation (NASA, 2004). Both these definitions are adopted by the SAMATE project. We understand that static analysis is useful to aid software safety and therefore is strongly related to software assurance.

In this research, the same terminology used by SAMATE (P. E. BLACK, 2011) will be used, in the sense that a software **vulnerability** is a property of a system’s security requirements, design, implementation, or operation that could be triggered, resulting in a security failure. A vulnerability is the result of one or more **weaknesses** in software requirements, design, implementation, or operation. This definition also means that a single weakness may or may not result in a vulnerability, depending on the software’s usage and execution. Whenever we use *bugs* or *flaws*, we mean weaknesses, and we will usually refer to them as flaws, as in SAMATE’s SATE V Ockham Sound Analysis Criteria analysis (Paul E. BLACK and RIBEIRO, 2016). This last work also states that static analyzers examine the source code of programs to find problems. In this sense, we use the term static analysis to reference the automatic search for problems in source code. A static analyzer receives source code as input and returns a report pointing to problems as its output.

The MITRE Corporation\(^4\) maintains the **Common Weakness Enumeration (CWE - Common Weakness Enumeration nodate)** (CWE), a community-developed dictionary of software weaknesses types with the support of several organizations, such as Apple, HP, IBM, and Microsoft. The CWE entries include, among other information, name and description of the weakness type, behavior of the weakness, description and consequences of the weakness exploit, and potential mitigation. Examples of software weaknesses registered as CWEs are buffer overflows, divisions by zero, and NULL pointer dereferences. Whenever possible, we will try to describe specific flaws matching them with CWEs. MITRE also maintains a dictionary for publicly known software vulnerabilities and exposures, the **Common Vulnerabilities and Exposures (CVE - Common Vulnerabilities and Exposures (CVE) nodate)** (CVE), which maps vulnerabilities in real-world software.

For SAMATE (P. E. BLACK, 2011), a **warning** is an issue identified by a static analyzer that might be a software flaw, and a **report** is defined as a set of warnings generated by a single run of a static analyzer in a software corpus. This terminology is important when referring to static analyzers and their outputs.

2.1.2 Handling Static Analysis Warnings

Previous studies show that the most relevant features for training accurate machine learning models to arbitrate about the positiveness of static analysis alarms are extracted from properties intrinsic to the analyzed project, namely the project change history, function and file names, and even the name of the programmer who introduced the change that triggered the alarm (KREMENEK, ASHCRAFT, et al., 2004; S. HECKMAN and WILLIAMS, 2009; JUNG et al., 2005; RUTHRUFF et al., 2008; YOON et al., 2014). Although these project-specific features are in great part responsible for the high accuracy of the models proposed

\(^3\)cnss.gov
\(^4\)mitre.org
up to now, a model trained on such features cannot be readily used to query about alarms generated for other projects, hampering the general availability of the model in automated post-analysis tools, which could support developers by decreasing the time spent inspecting false alarms.

**Boogerd and Moonen, 2006** present a warning ranking technique that uses the probability of a code section being executed to prioritize warnings. The authors claim that other than using the warnings severity and the Z-ranking technique, they were not aware of any other methods to rank warnings by the time of their study. Z-ranking (**Kremenek and Engler, 2003**) is a technique to rank warnings using the frequency counts of true and false positives. It was validated against random ranking algorithms. The authors of the Z-ranking technique emphasize that a classification system cannot be perfect since static analysis cannot be perfect itself. We also use a random ranking algorithm to assess our warnings ranking approach.

Later, **T. B. Muske et al., 2013** pointed out the difficulties in reviewing a high number of warnings and divided the warnings into different categories to identify redundant warnings and estimate which categories were more prone to hold false positives. This work also mentioned that if a code is never executed, warnings pointing at it should have lower review priority. We divide warnings into categories and use these categories as one of the features to train the classification model used to rank warnings.

**Ruthruff et al., 2008** propose a method to maximize the return on investment of static analyzers to predict if a warning is an actionable fault, i.e., if it is not a false positive and if a programmer should fix it. It uses a screening approach for model building that discards metrics with low predictive power. Among the factors used to predict false positives, which happened 85% of the times, are the priority given by the static analyzer and the file length and indentation, meaning that, differently from the Z-ranking approach, the authors did run some static analysis in the code just to extract factors from it. The present study proposes a method to rank warnings based only on the information provided by the analyses.

In their work, **Spacco et al., 2006** implement a technique to track static analyzers warnings that correspond to the same flaw across different software versions. This technique would enable the assembly of tool report deltas, helping developers to focus only on new flaws. Since false positives are not usually removed from one software version to another, this approach could help to identify them. By implementing similar techniques to track warnings across versions in our continuous static analysis tool, we could to present static analysis reports containing only the warnings introduced in specific software versions, diminishing the efforts of analyzing tool reports. These reduced reports might be useful to reduce the number of false positives shown to software developers since false positives tend to remain across different versions of a software package. Authors claim that although it seems an easy task, tracking warnings across software versions is not a trivial problem. Hence, we leave the technique implementation for future work, as pointed in Chapter 6.

**Kim and Ernst, 2007** mention the high rates of false positives in static analysis and claim that sometimes static analyzers prioritize their warnings inefficiently. Later, they propose a method to prioritize warnings by searching for bug fixes in the software change history. They then assume that if a warning from a specific category was removed by a fix, then all warnings in that category are important. By improving the precision of the tools used for the study on finding true positives, the authors showed that the software change history may be important to prioritize warnings.

In another work, **Penta et al., 2008** performed an analysis of the evolution of vulnerabilities detected by static analyzers in three different open source applications: Samba, Squid, and Horde. The authors analyzed how the number of vulnerabilities varies
over time. Since they analyzed different development versions (not only releases), they could investigate aspects of the development process, like bug fixing efforts right before a release. The work aimed to understand how long vulnerabilities tend to remain in the system by modeling the vulnerabilities decay time with a statistical distribution and comparing the decay time of different classes of vulnerabilities.

Our study differs from the ones cited above by assessing static analysis warnings only with the information present in the warnings themselves, as discussed during the introduction of this paper (Chapter 1). This difference makes predicting whether a warning is an actual bug or a false positive harder, since related works emphasize that the most important characteristics to do this are internal to the analyzed project. To compensate for this, we use multiple static analyzers to generate more information and correlate the information provided by them to better assess the correctness of a given warning. Since this strategy still might result in a low-quality classifier, we turn to ensemble techniques to generate and combine multiple weak classifiers generated this way into a stronger one (Russell and Norvig, 2003).

Xypolytos et al., 2017 proposed an approach to prioritize the output of multiple static analyzers by assigning confidence scores to the tool warnings based on how well each tool handles a specific flaw category. Although one may compare the authors’ approach to the one presented here since it also assesses the warnings solely based on the information present in the warnings themselves, the study is in its initial stages and the authors did not provide statistically significant results.

Next, we introduce the continuous static analysis system developed during this work, describe its features and discuss the reasons behind the technical decision taken during the platform development.
Chapter 3

Continuous Static Analysis Tool Design

In Free/Libre/Open Source Software (FLOSS) projects, a common source of bug reports are the GNU/Linux distributions. These distributions ship thousands of software projects, which they call packages. Distribution developers refer to the projects that maintain the software they ship in the distribution as upstream. It is not unusual for distribution developers to report bugs in upstream projects or to send patches to fix bugs found by their distribution users or during the packaging process. Figure 3.1 shows a typical bug reporting workflow, where users report the bugs they find to the distribution developers, who in turn report these bugs (in addition to bugs found during the packaging process) to the upstream project maintainers of the package containing the bugs.

By continuously running multiple static analyzers in several of these packages, i.e., once for each version of the package, we can create a database of static analysis reports on software projects of different sizes and application domains. Developers can then use the information in this database to find and act on software flaws.
We chose to use GNU/Linux distributions due to the high amount of software packages available and the well-defined and documented interfaces they provide to download the latest versions of these packages. It is also an advantage that the cultural norm for GNU/Linux distribution developers is to report (and often propose fixes for) bugs. Therefore, using their repositories for this work may provide a broader user base for the tools and techniques developed in this research. Hence, kiskadee was designed with GNU/Linux distribution developers as its target audience, which could also be expanded to upstream software developers and quality assurance teams.

### 3.1 Features and System Overview

To continuously run static analysis on software packages and handle the false alarms generated by these analyses, we developed *kiskadee*. Its main features are described below:

1. **Monitor specific software repositories for new releases.** This way, we can track and analyze each release of the monitored packages since we started monitoring them.

2. **Download new software packages from the repositories.**

3. **Run multiple static analyzers on the downloaded packages.**

4. **Translate each static analyzer report to a common report format.** This is needed to enable analyzing the reports during the ranking and visualization phase. Since the system is extensible, we do not want to write parsing rules for each static analyzer report format to compare the analyzers, so we do this before saving the static analysis output.
5. **Rank warnings.** The warnings are ranked based on the probabilities assigned by our classification algorithm of each warning being a true or false positive. The methodology and techniques applied to implement this feature are described in Chapter 4.

6. **Save the reports in a database.**

7. **Make data available through an API.**

8. **Present data to users on a web user interface.** Users can filter warnings per package name and version.

Figure 3.2 represents kiskadee’s architecture overview, where the numbers denote its execution flow.

- In steps (1) and (2), kiskadee monitors software repositories for new releases.
- In step (3), kiskadee downloads the source code of each new software version in a repository it monitors and runs a set of predefined static analyzers on it in step (4).
- In step (5), kiskadee translates each static analyzer report to a common warnings report format. This common format is needed because each static analyzer defines its unique format to report warnings.
- In step (6), kiskadee ranks the warnings based on their probability of being real bugs, where warnings on the top have a higher probability of being real bugs and warnings on the bottom are more likely to be false positives. To perform this ranking step, we use a classification model, described in Chapter 4.
- The ranked warnings are then saved in a database in step (7), using kiskadee’s common warning report format.
- Finally, in step (8), kiskadee provides an API consumed by a visualization tool to display the ranked warnings filtered by package versions. The information provided can be used either by distribution developers to evaluate and report possible bugs upstream or by upstream developers themselves. Researchers can also use kiskadee’s database in different contexts.

The next sections describe important details related to kiskadee features. Then, in Section 3.6, we discuss kiskadee’s architecture and implementation details. We finish the chapter by discussing software projects related to kiskadee in Section 3.7.

### 3.2 Monitoring Software Repositories

Although pointing kiskadee to GNU/Linux distribution repositories may have advantages, most distributions do not package every single new software version for the packages it distributes. Different reasons lead to the non inclusion of some software versions in distributions. For instance, some software projects claim that they only provide stable software every other release. Other projects release versions too fast, making the effort of packaging every single release pointless to the distribution packager.

The Fedora Project uses an external system to monitor the software projects they distribute. Anitya ([Red Hat](http://redhat.com), nodate) maps upstream projects to distribution package names. Whenever a new version of an upstream project is released, Anitya publishes the new release in a Fedora infrastructure publish-subscribe system where other systems in the
distribution infrastructure can handle it. New software versions are published by Anitya as soon as they are released upstream. Analyzing these new software versions with kiskadee as early as they are released allows the distribution developers to address potential bugs found by kiskadee before the software is shipped to final users. It also ensures that every new monitored software release gets analyzed, even if the new version is not packaged in the distribution, for the reasons aforementioned. Therefore, kiskadee monitors packages by reading information published by Anitya in the Fedora infrastructure publish-subscribe system.

kiskadee can monitor other software repositories as well through its plugin architecture (kiskadee’s fetchers). Each fetcher must implement functions that define which repository to monitor, how to monitor it, and which static analyzers to run for that repository. Hence, we can extend kiskadee to run different static analyzers for different software repositories or GNU/Linux distributions.

### 3.3 Static Analyzers

The NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project provides a list of source code security analyzers, which they define as static analyzers that examine source code to detect and report weaknesses that can lead to security vulnerabilities.

We ran kiskadee with three static analysis tools to generate the warnings data set for our experiments. Based on SAMATE tools list and references, the criteria to select the tools were:

1. **The tools must be able to examine C/C++ code for security flaws** (e.g., buffer overflows, null pointer dereferences). Different programming languages have different problems, and we do not want to address this variable at the moment. Since C and C++ programming languages are often used to develop critical software systems and several software flaws are mapped under Common Weakness Enumeration (CWE) for these languages, we chose to work with these languages.

2. The tool source code must be released under a FLOSS license.

Criterion 1 ensures the tool can analyze a subset of the test cases in our data set, introduced in Section 4.1, whereas criterion 2 preserves us from disputes by tool vendors regarding the analysis of the results (such as allegations of sub-optimal tool calibration or detrimental calculation methodology). FLOSS tools also simplify the process of retrieving string constructs for static analyzer warning messages and categories when necessary, since we can verify their source code.

Following the criteria, the static analyzers selected were Clang Static Analyzer (version 3.9.1), Cppcheck (version 1.79), and Frama-C (version 1.14 with the value analysis plugin activated).

Frama-C with its value analysis plug-in was the first choice since it was the tool used for the Ockham Criteria (Paul E. Black and Ribeiro, 2016) and the authors claim Frama-C is a sound tool, not producing false positives, which may help on the ranking phase of this study. It uses abstract interpretation techniques. Although SAMATE does not list Frama-C in its recommended tools list, it is recommended by the Flawfinder website, one of the secondary

---

1. [samate.nist.gov/index.php/Source_Code_Security_Analyzers.html](http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html)
2. In fact, if we chose to work with proprietary tools, we would need to contact the vendors about our experiments and request permission before publishing our results.
3. [dwheeler.com/flawfinder](http://dwheeler.com/flawfinder)
tool lists recommended by SAMATE. Next Cppcheck was chosen, which is a static analyzer that works by matching on tokens, without more in-depth analysis methods. Cppcheck authors also claim to avoid generating false positives. Finally, Clang Static Analyzer works by performing inter-procedural analysis on the source code and, differently from the other two selected tools, the authors do not claim that this tool does not generate false positives.

Table 3.1 shows the tools integrated into kiskadee. The same tools were also employed to generate our experimental data set, as we discuss in Chapter 4.

| Tool               | Target Language | License  | Version |
|--------------------|-----------------|----------|---------|
| Clang Static Analyzer | C/C++          | NCSA/MIT | 3.9.1   |
| Cppcheck           | C/C++          | GPLv3    | 1.79    |
| Frama-C*           | C              | LGPLv2   | 1.14    |

* Although Frama-C cannot analyze C++ code, it meets criterion 1 by analyzing a substantial subset of Juliet test cases. Frama-C was used with its value analysis plugin activated.

**Table 3.1: Selected static analysis tools**

We designed kiskadee in an extensible fashion. Whenever the desire to integrate new static analyzers to it arises, the effort needed to do so is decoupled from kiskadee source code. Instead, one would just need to extend the external common reporting language library used by kiskadee to parse and convert the target static analyzer report to the common report. We now, we discuss this common reporting language.

### 3.4 Common Reporting Language

A previous work has discussed the problem of having a common reporting language for static analyzers. By applying the Ockham Sound Analysis Criteria (Paul E. Black and Ribeiro, 2016), we designed a simplified common bug report format to compare known bugs with bug reports from the tools evaluated with the criteria. The criteria data and programs to reproduce results are available in the SATE V Ockham Sound Analysis Criteria NIST Internal Report (Paul E. Black and Ribeiro, 2016), where the bug report formats specified can be retrieved from and replicated. While the format was fit to perform the analysis, we designed it on top of the criteria concepts and the tools used for the criteria, which needed the following information:

- a specific location for the bug;
- the type of the bug being reported (related to a CWE);
- a code snippet with the buggy code;
- information to identify false alarms on that code snippet.

We defined an XML format to extract information from the used test suite (code snippets with known flaws injected) to verify the soundness of the analyzed tools. Listing 3.1 is an example of the format, extracted from one of the files with Ockham results. Although tools to extract information from the format were developed and are available, no formal definition of the format was described, since this format was used for this specific version of the criteria.
To compare the findings of the evaluated tools with actual data from the test suite, we parsed the tool reports and converted them to a straightforward comma-separated values (CSV) format containing file name, line and bug category related to a CWE. An example of this CSV format, extracted from the Ockham Criteria results, is shown in Listing 3.2.

```plaintext
CWE121__char_type_overrun_memcpy_01.c, 43, divByZero
CWE121__char_type_overrun_memcpy_01.c, 62, divByZero
CWE121__char_type_overrun_memcpy_02.c, 45, divByZero
CWE121__char_type_overrun_memcpy_02.c, 73, divByZero
CWE121__char_type_overrun_memcpy_02.c, 92, divByZero
CWE121__char_type_overrun_memcpy_03.c, 45, divByZero
CWE121__char_type_overrun_memcpy_03.c, 73, divByZero
CWE121__char_type_overrun_memcpy_03.c, 92, divByZero
CWE121__char_type_overrun_memcpy_04.c, 51, divByZero
CWE121__char_type_overrun_memcpy_04.c, 79, divByZero
```

Code Listing 3.2: Ockham Criteria CSV bug report example

To rank warnings, we would like to have information on analyzed software versions, which static analyzer performed the analysis, the version of the analyzer, the severity of the problem found, and the string message given by the analyzer. All this information may be interesting for research or bug reporting purposes. Thus, if we were to use the format we developed for the Ockham Criteria, changes would be needed, both to the format and to the tools provided to work with it.

FLOSS development communities have also been discussing a common static analysis report output format. The Fedora Project Static Analysis Special Interest Group (SIG) ([fedora:static:sig](https://fedora:static:sig)) designed a tool to run static analyzers during the package build process ([fedora:mockwithanalysis](https://fedora:mockwithanalysis)). Although the tool itself is in its early development stages and not ready for usage in production, the developers discussed a common report format for the static analyzers in their mailing lists ([fedora:mlist](https://fedora:mlist)). After a few iterations, with Debian Project developers collaboration, they created and evolved Firehose, a complete definition of a common warnings report format for static analyzers and a set of tools to generate, parse, and verify this format. Firehose addresses the issues previously discussed, as analyzer name and version, optional fields for severity, and complete warning messages. It also allows custom fields in the format, which we use to include kiskadee’s ranking information. Appendix B is a RELAX-NG schema describing the firehose format. We use Firehose as kiskadee’s common warning report format, eliminating the need to design a new format and to develop the tools to handle it, like parsers and generators.

Moreover, as previously mentioned in Section 3.3, if we want to integrate new static
analizers to kiskadee, this must be performed in Firehose, by implementing a new parser for the target static analyzer. During the Google Summer of Code 2017\(^4\) program, we extended Firehose to parse both the Frama-C static analyzer, which we use for our experiments described in Chapter 4, and the Flawfinder static analyzer.

### 3.5 API and Web User Interface

To provide the static analysis reports and warnings ranking data to external applications, kiskadee exposes an API. The API responses contain the following information:

- List of all active fetchers running in kiskadee’s current instance.
- List of all packages analyzed by kiskadee up to the time of the API request.
- List of each version analyzed for a given package.
- The full analysis report for each package version analyzed, in the Firehose format. Also includes ranking information.

Provided with this API and aided by students from the University of Brasilia (UnB), we developed a user interface (UI) for kiskadee. We based many of the UI ideas on concepts previously discussed by the Fedora Static Analysis SIG on the project development mailing lists. Proper credits and acknowledgements for the UnB contributions can be found in the Acknowledgements section in the beginning of this document.

Figure 3.3, Figure 3.4, and Figure 3.5 show some screens of the UI developed by the students. The next section presents more specific details on kiskadee’s implementation.

![Figure 3.3: kiskadee UI: Package search page](image)

### 3.6 Architecture and Implementation Details

Kiskadee’s architecture aims to decouple source code analysis from software packages release monitoring. To accomplish that, we have separated the modules responsible for fetching package versions and source code, and the module responsible for running the analyses. The communication between them is done through queues, as shown in Figure 3.2 by A, B, and C.

The packages_queue (A) is used by the fetchers to enqueue the packages that should be analyzed. The monitor module consumes this queue to check if the enqueued package was already analyzed. The second queue, named analysis_queue (B), is consumed by the runner module to receive packages that must be analyzed, which are enqueued by the monitor module. If a dequeued package does not have an analysis instance in the database,

\(^4\)https://summerofcode.withgoogle.com/archive/2017/projects/5729359134457856/
the monitor module enqueues it for analysis in the analysis_queue. When an analysis is performed, the runner module sends the analysis report to the monitor module, by enqueuing it in the results_queue (C). Some package metadata and the analysis are then saved in the database by the monitor module.
The fetchers module implements a system to easily define targets to be monitored by kiskadee. A target is a software repository monitored for new packages. Whenever kiskadee is loaded, it loads all fetchers installed in a specific path. From that moment on, kiskadee starts monitoring all the targets defined by the loaded fetchers.

A new fetcher must inherit from kiskadee’s Fetcher and implement some required abstract methods so kiskadee knows how to

- monitor the target;
- download source code; and
- compare target’s software versions.

Listing 3.3 shows a complete example of a custom kiskadee fetcher implementation.

```python
import kiskadee.queue

class Fetcher(kiskadee.fetchers.Fetcher):
    """Example kiskadee fetcher implementation."""

    def get_sources(self, source_data):
        """Use sources from test suite."""
        return '/quotesingle.Varkiskadee/ tests/test_source/test_source.tar.gz'

    @kiskadee.queue.package_enqueuer
def watch(self):
        """There is no proper API to inspect new example versions."
        It should not matter, since example will not receive updates."
        example = {}
        example['fetcher'] = kiskadee.fetchers.example.Fetcher()
        example['version'] = '0.1'
        example['name'] = 'example'
        return example

    def compare_versions(self, new, old):
        """Compare package versions."
        """
        Example has only one version. Let's just pass.
        """
        return 0

Code Listing 3.3: Example kiskadee fetcher
```

3.7 Related Projects

A few different software projects overlap with certain kiskadee features. We finish this chapter by listing and discussing some of these software projects.

3.7.1 Debile

Debian Project has been developing Debile, a reduced Debian build system to aid developers to build packages or run custom tools against source or binary packages. Debile
can run multiple static analyzers in Debian repository packages and return the reports in the firehose format (https://github.com/opencollab/debile). While the tool is still in early development (not in production), kiskadee design was inspired in Debile, with some simplifications and abstractions, allowing us to fetch source code from different software repositories. While Debile aims to check the quality of Debian packages, kiskadee focuses on a broader scope, being distribution agnostic and fetching software packages from the upstream projects.

3.7.2 mock-with-analysis

Similar to Debile, mock-with-analysis (https://github.com/fedora-static-analysis/mock-with-analysis) is a command line tool to build RPM\(^5\) packages in a clean chroot and run multiple static analyzers on the source code of the package being built. This tool is based on mock\(^6\), a tool used by Fedora developers to build RPM packages in a clean environment with a minimum set of packages pre-installed. While mock-with-analysis is just a prototype at the moment and is not fit for general usage, it inspired the development of Firehose, which is an essential component for kiskadee.

3.7.3 Coala

Coala (https://coala.io) provides a single interface to run multiple static analyzers in software packages written in numerous programming languages and producing a report in a common reporting language defined by the Coala project. The tool is not restrained to security oriented static analysis, it also runs linters and other analyzers to check code quality and style.

While all the projects mentioned above are interesting projects, and Coala being a production-level open source software sponsored by large companies, none of them proposes an approach to reduce the number of false positive warnings generated by the analysis, which increases when running multiple static analyzers in a software package. In the following chapter, we present our approach to handling these false positives by training a classification model and ranking the warnings based on classification probabilities.

\(^5\)rpm.org

\(^6\)https://github.com/rpm-software-management/mock
Chapter 4

A Static Analysis Warnings Ranking Model

We approach the problem of ranking multiple static analyzer warnings with a supervised learning method to arbitrate about the warnings positiveness. To learn from observed examples, we need to obtain and process a data set of static analysis warnings labeled as true or false positive. We describe the steps taken in this direction in the next section.

4.1 Data Extraction

To build a predictive model as discussed, we need to obtain good and sufficient data to train it. The first step we take in this direction is choosing both the source code to analyze and the tools with which to analyze it. Then, to adequately process the output of multiple tools, we need to normalize the information provided by them. We approach this in two levels: at the first level, we translate alarms from the different tools into a unified format and apply labels to each warning to distinguish them into either true positives or false positives. The labeling step is feasible because we use a set of known software flaws as input for the static analyzers, as we describe in the next section. At the second level, we associate each warning to specific features that will compose a data set to train our prediction model.

4.1.1 Choosing Data Sources

A data set of labeled static analysis warnings may be obtained by running static analyzers on previously selected source code and matching the triggered warnings with actual software defects, labeling the warnings as true or false positives. The source code used for extracting the data set may consist of real-world software or synthetic test cases, i.e., programs written with intentional defects.

Previous studies have used real-world software to arbitrate about the positiveness of static analysis warnings (Kremenek, Ashcraft, et al., 2004; Jung et al., 2005; Yoon et al., 2014; Yüksel and Sözer, 2013; Ruthruff et al., 2008). A relevant obstacle for using real-world software to retrieve a data set is that one must meticulously inspect each alarm triggered by a static analyzer and the relevant location in the analyzed program to determine whether the alarm is a true or a false positive. Unless one has access to an environment where alarms positiveness information is already available, as in Ruthruff et al., 2008, performing these manual inspections is an overly time-consuming approach, which must be performed for each different static analysis tool being assessed. Differently, the location and other details of the injected flaws in synthetic test cases are previously
22

known, facilitating studies and freeing researchers from performing expensive searches along their data sets.

We use Juliet (Boland and P. E. Black, 2012) version 1.2 to generate our data set. Juliet is a synthetic C/C++ test suite created by the United States National Security Agency (NSA) and distributed by the National Institute of Standards and Technology (NIST) under the public domain. It is composed of 61,387 test cases covering 118 different software security-related flaw categories. Each test case is a code section with an instance of a specific flaw (to catch true positives) and an additional section with a correct, fixed instance of that previous flawed section (to catch false positives). Juliet also includes a user guide with instructions on how to assess and label static analysis tool warnings generated over its test cases.

We ran three static analysis tools to generate the data set for our experiments. Here we use the same static analyzers integrated into kiskadee since their output will compose kiskadee final static analysis reports, which we want to rank with our model. While the Criteria employed to select the tools are described in Section 3.3, the list of selected static analyzers is shown in Table 3.1.

After choosing Juliet and the analysis tools, we must consider how to combine them to build our data set. Let \{J\} be the whole set of Juliet test cases and \{T_x\} the subset of Juliet test cases that tool \(x\) is able to detect. Figure 4.1 shows a Venn diagram for the situation where three static analysis tools are employed. We may choose to process \{J\}, \{T_1 \cup T_2 \cup \ldots T_n\}, or \{T_1 \cap T_2 \cap \ldots T_n\}. We may even imagine processing, with each tool \(x\), only the subset of test cases \{T_x\} that the tool is able to detect.

\[T_1 \cap T_2 \cap \ldots T_n\]

\[J\]

\[T_3\]

\[T_1\]

\[T_2\]

Figure 4.1: Juliet test suite universe (\(J\)) and its subsets detected by each static analyzer \(T_x\).

In our case, there is no difference between the first and second options: since we are dealing with true and false positives only, and not negatives, \(J - (T_1 \cup T_2 \cup \ldots T_n)\) (region represented in Figure 4.2) is never included in the results. There might be some difference, however, in studies focusing on negatives.

Similarly, there is no difference between processing different test subsets with each tool and processing \(T_1 \cup T_2 \cup \ldots T_n\) (Figure 4.3): test cases that cannot be addressed by some tool will not be in both cases. Again, focusing on negatives would invalidate this, hence it is important to reinforce that we only care about positives and false positives for our
4.1 | DATA EXTRACTION

Figure 4.2: Venn diagram for \( \{J = (T_1 \cup T_2 \cup \ldots T_n)\} \) (grey area), where \( J \) is the Juliet test suite universe and \( T_x \) is the coverage of the static analyzer \( x \).

Figure 4.3: Venn diagram for \( \{T_1 \cup T_2 \cup \ldots T_n\} \) (grey area), where \( J \) is the Juliet test suite universe and \( T_x \) is the coverage of the static analyzer \( x \).

The only truly different approach, therefore, is to analyze only \( \{T_1 \cap T_2 \cap \ldots T_n\} \) (region represented in Figure 4.4). This has several drawbacks. The model we want to build works by identifying correlations among data points. With this choice of subset, we would exclude data in \( \{(T_1 \cap T_2) - T_3\} \) etc., shown in Figure 4.5, reducing the strength of the algorithm. This would also prevent us from using the correlations obtained to rank warnings thrown by a single tool. Since ranking is a very important aspect of our work, this would be highly inappropriate.

In light of the previous discussion, we consider the use of \( \{J\} \) as the simplest approach. As discussed, this does not mean we are adding false or noisy data into the model: some
Figure 4.4: Venn diagram for \( \{T_1 \cap T_2 \cap \ldots \cap T_n\} \) (grey area), where \( J \) is the Juliet test suite universe and \( T_x \) is the coverage of the static analyzer \( x \).

Figure 4.5: Venn diagram for \( \{(T_1 \cap T_2) - T_3\} \) etc. (grey area), where \( J \) is the Juliet test suite universe and \( T_x \) is the coverage of the static analyzer \( x \).
4.1 | DATA EXTRACTION

data points may be more or less relevant to the classifier, depending on which and how many tools brought them up, but identifying this relevance is exactly the role of the classifier.

However, before examining Juliet with the selected static analysis tools, we still prune the test suite to prevent analysis of test cases that depend on constructs of specific operating systems or external libraries. Table 4.1 shows the total number of test cases in Juliet before pruning warnings and the number of test cases after the pruning step for both C and C++. The latter are the tests examined by the static analyzers for alarms generation, consisting of 39,100 C/C++ test cases.

|             | Before Pruning | After Pruning |
|-------------|----------------|---------------|
| C           | 36,078         | 22,459        |
| C++         | 25,309         | 16,641        |
| Total       | 61,387         | 39,100        |

Table 4.1: Number of Juliet test cases

4.1.2 Collecting Labeled Warnings From Multiple Static Analyzers

Based on Juliet 1.2 documentation, we process each file in the remaining test cases to produce a list (L) with information on whether a static analysis warning for a given location should be labeled as true positive or false positive. If a static analyzer generates an alarm for a location not included in this list or for a category not covered by the flaws present in that location, we do not draw any conclusions about such location, since it would require thorough manual inspection. Instead, we discard that warning. Then, we run each static analyzer on the pruned test suite to generate the static analyses reports. Table 4.2 shows the total number of warnings generated, before discarding the warnings whose labeling step cannot be automated.

| Tool               | Warnings |
|--------------------|----------|
| Clang Static Analyzer | 37,229   |
| Cppcheck           | 124,025  |
| Frama-C            | 120,573  |
| Total              | 281,827  |

Table 4.2: Total number of warnings generated per tool

Seeing that each tool has its unique report format, we reduce the effort of parsing reports by converting them all to a unified report (UR) in a common format. Here, we use the Firehose format, introduced in Chapter 3. By matching each possible warning produced by the static analyzers with the corresponding flaw categories covered by Juliet, we can use the list L to produce labels for each warning in UR.

The set of labeled warnings can finally be examined to have a training set extracted from it, as discussed in the next section. Table 4.3 summarizes the findings of the static analyzers on the test cases, including the number of true and false positives generated (TP and FP, respectively), the false positive rates, and the precision for each tool and for UR, the aggregated reports composed of the warnings of all tools.
4 | A STATIC ANALYSIS WARNINGS RANKING MODEL

| Tool            | Warnings | TP  | FP  | FP Rate | Precision |
|-----------------|----------|-----|-----|---------|-----------|
| Clang Analyzer  | 6207     | 984 | 5223| 0.84    | 0.16      |
| Cppcheck        | 4035     | 314 | 3721| 0.92    | 0.08      |
| Frama-C         | 15717    | 8892| 6825| 0.43    | 0.57      |
| Aggregated tools| 25959    | 10190| 15769| 0.61    | 0.39      |

Table 4.3: Labeled warnings per tool

4.1.3 Extracting Features from Labeled Warnings

We obtain the features used to train our classifier from the set of labeled warnings. Here, a feature is any characteristic that can be attributed to a warning in the data set. To select relevant features for false and positive alarm classification, we refer to previous studies that also rely on characteristics extracted from alarms and source code to classify alarms (Ruthruff et al., 2008; Kremenek, Ashcraft, et al., 2004; S. Heckman and Williams, 2009; Jung et al., 2005). For instance, Kremenek, Ashcraft, et al., 2004 demonstrate that the tool warning positiveness is highly correlated to code locality. However, we only use the data extracted from the warnings themselves, not relying on other information, such as the analyzed project change history, size and complexity metrics, or artifact names (e.g., file name, function name, package name). By not collecting such information, we can produce a more generic model applicable to different contexts without the need to perform the expensive training step for each project one may want to analyze.

We extract our set of features by processing the aggregated report of labeled warnings. While we can infer the name of the tool that triggered a warning, the programming language analyzed, and the severity of the warning by looking at a single warning at a time, other features require processing the whole aggregated report to be extracted. Namely, these are (1) the number of times the same location was pointed as flawed in the report, (2) the number of warnings triggered around the location of a given warning (e.g., warnings for locations at most 3 lines away from the current warning), (3) the category of the software flaw suggested by the warning, (4) which other static analyzers generated warnings for the same location, and (5) the number of warnings generated for the same file the current warning is pointing to.

Except for warning category and warning severity, the features mentioned above can be readily obtained for a given warning by analyzing the whole aggregated report. We assign a category to a warning by matching its message with one of the categories in Table 4.4. Since Juliet test cases cover a wide range of software flaw categories and each static analyzer can detect a specific set of bug categories (necessarily mapped to Juliet categories), this list can be extended, breaking categories into more specific subcategories to seek improvement in the importance of this feature in the prediction model.

To assign a severity to the warnings, we normalize the severities of each tool to values between 0 and 5. We assume that 5 refers to errors in the source code, like syntax errors that would prevent the source code to be compiled; 4 refers to potential flaws; 3 and lower values refer to style or similar issues. Thus, If a tool does not provide severity information, we assume 4. The following list summarizes the features obtained from the labeled aggregated report.

- **Tool name**: Name of the tool that generated the warning;
- **Number warnings in the same file**: Number of warnings found in the same file as the warning in question, regardless of what tool generated it;
4.2 Arbitrating on Warnings Positiveness

Given the data collected using the methodology presented in the previous section, we now build a prediction model to classify each triggered warning as being a true positive or a false positive. The classification results may also be used to rank the warnings, as we describe in the second part of this section.

4.2.1 Training Decision Trees with AdaBoost

To make our model applicable to a broader variety of projects in a generic fashion, we only use data extracted from the warnings themselves. Since we do not inspect the source code or project history, which are shown to be the best places to look for features when classifying source code static analysis warnings as true or false positives, we turn to ensemble learning methods to train several weak classifiers with our feature set. These weak classifiers combined can then arbitrate on new examples together, composing a stronger classifier.

A weak classifier performs slightly better than a random classifier, i.e., \( \epsilon < 0.5 \) where \( \epsilon \) is the classification error of the classifier. Ensemble learning methods combine the prediction of weak classifiers to build stronger classifiers with lower errors (Russell and Norvig, 2003). One widely used ensemble learning method is boosting (Russell and Norvig, 2003), whose main idea is to run a weak learning algorithm several times in different distributions of the training set to generate various weak classifiers to be combined into a stronger one. For

| Category name | Description |
|---------------|-------------|
| buffer        | Buffer related issues (e.g., Buffer overrun) |
| overflow      | Integer overflow and underflow |
| div0          | Divisions by zero |
| uninitvar     | Uninitialized variable |
| unusedvar     | Unused variable |
| pointer       | Pointer issues (e.g., Null pointer dereference) |
| operator      | Misused operators (e.g., \( if(myVar = buf[i]) \{ \} \)) |
| funcparams    | Issues with function parameter |
| alwaysbool    | Expression is either always true or always false |
| memory        | Memory issues (e.g., Memory leak, Double free) |
| other         | Warnings not fit to the categories above |

Table 4.4: Possible Assigned Warning Categories

- **Category**: Category of the warning, as shown in Table 4.4;
- **Redundancy level**: Number of warnings triggered for that same line, regardless of what tool generated it;
- **Number of neighbors**: Number of warnings less than 4 lines away from the triggered warning. Does not include the ones counted in Redundancy level;
- Finally, we include one **boolean feature** for each of the static analyzers to indicate if that tool triggered a warning for that location. We understand that these features help confirm software flaws and take advantage of the fact that each tool performs differently for specific cases.
this study, we chose to use the AdaBoost algorithm (Freund et al., 1999), a more general version of the original boosting algorithm (Polikar, 2006).

AdaBoost works by calling a base learner algorithm (\( h \)) repeatedly (\( T \) times) while maintaining a set of weights over the training data set. The weights are used to select a subset of the training data set to serve as input for the base learner algorithm in a given iteration of AdaBoost (higher weights mean a higher likelihood of an example being selected for an iteration). Each example in the training data set starts with the same weight, but they are updated after each iteration: the weights of misclassified examples are increased and the weights of examples correctly classified are decreased. This forces the base learner algorithm to focus on the hard examples of the training data set (Freund et al., 1999). Each weak classifier is assigned a voting weight (\( \alpha \)) based on its importance. The strong binary classifier (\( H \)) is given by Eq. (4.1), where \( x \) is a new example to be classified.

\[
H(x) = \text{sign}\left( \sum_{t=1}^{T} \alpha_t h_t(x) \right) \tag{4.1}
\]

The AdaBoost algorithm works with any given base learner. We use a decision tree learning algorithm as our base learner because we have both categorical and non-categorical features in our data set, as shown in Section 4.1, and decision trees can work with both, without the need to pre-process the data set. Furthermore, as shown in the literature, decision trees perform well with AdaBoost (Drucker and Cortes, 1996). Below, we present a complete definition of the AdaBoost algorithm as defined by Freund et al., 1999.

**Algorithm 1** AdaBoost algorithm (Freund et al., 1999)

Given: \((x_1, y_1), \ldots, (x_m, y_m)\) where \(x_i \in X, y_i \in Y = -1, +1\)

Initialize \(D_1(i) = 1/m\).

for \( t = 1, \ldots, T \) do

Train weak learner using distribution \(D_t\).

Get weak hypothesis \(h_t : X \rightarrow \{-1, +1\}\) with error

\[
\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i].
\]

Choose \(\alpha_t = \frac{1}{2} \ln \left( \frac{1-\epsilon_t}{\epsilon_t} \right)\).

Update:

\[
D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} 
  e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\
  e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i
\end{cases}
= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}
\]

where \(Z_t\) is a normalization factor (chosen so that \(D_{t+1}\) will be a distribution).

end for

Output the final hypothesis:

\[
H(x) = \text{sign}\left( \sum_{t=1}^{T} \alpha_t h_t(x) \right)
\]

We divide our data set (the list of labeled features obtained in Section 4.1) into a training
set and a test set. The training set is built by randomly selecting 75% of the examples labeled as true positives and 75% of the examples labeled as false positives from the features data set. We then proceed to train our predictive model using 10-fold cross-validation with the training set. This cross-validation consists in dividing a data set into 10 groups: 9 groups are used to train the model while the remaining group is used to test it. The training step is repeated ten times, switching the group used for testing until all groups were used in this way. We perform the 10-fold cross-validation technique with different values for $T$ in AdaBoost. We then compare the average performance of the classifiers obtained for each different value of $T$ validated in this manner and use the best model trained during the cross-validation for that $T$ to classify the test set.

4.2.2 Ranking Static Analysis Warnings

We use the model trained as described in this chapter to rank the warnings in a static analysis report based on the model classification probabilities. We reorder the warnings in a list according to the probability given by our classification model of the warning being a true positive, where warnings with higher probabilities are ranked in the top of the list and warnings with lower probabilities of being true positives are arranged in the bottom of the list. This way, a programmer inspecting the ranked static analysis report may examine only the top warnings in the list up to a given threshold, assuming a certain risk of missing true positives. Alternatively, he/she may stop inspecting warnings when false positives start to abound.
Chapter 5

Results and Discussion

After the steps of choosing the input data, selecting the static analysis tools, and processing their reports to generate a predictive model, the question is how well does it work? We now discuss the performance of the binary prediction model trained with AdaBoost to classify static analysis tool warnings as true or false positives and evaluate the ranking system described in Section 4.2.2.

5.1 Classifier

The data collection steps of our experiment resulted in a data set of 25,959 labeled warnings. To train the classifier, we divided them into a training set with 19,470 examples and a test set with 6,489 examples. The former was used to train the model and tune parameters with 10-fold cross-validation.

The final classifier was trained with 100 AdaBoost iterations ($T = 100$), producing 100 decision trees to compose the predictive model, reaching 0.8 mean accuracy (the proportion of correct classifications in relation to the total input) in the cross-validation during training. This value for $T$ was selected based on the performance of the mean accuracy of the model during training for 10-fold cross-validation, as can be seen in Table 5.1.

| Number of trees ($T$) | Mean accuracy |
|-----------------------|---------------|
| 10                    | 0.797         |
| 50                    | 0.791         |
| 100                   | **0.801**     |
| 150                   | 0.794         |

*Table 5.1: Mean training classification accuracy (cross-validation)*

Accuracy is just one of the interesting metrics we can derive from the experiments. Table 5.2 presents a confusion matrix for the classification of the examples in the test set with our trained prediction model. It readily shows the low number of misclassified labeled true positives (i.e., our model classified real bugs correctly most of the time) and the high number of misclassified false positives (our model classified false bugs incorrectly more often than desirable) but, most importantly, we can use it to calculate the classification accuracy, precision and recall for the model over the test data set.

Accuracy (0.805) is the number of correct classifications (True Positive and True Negative) divided by the total input (all generated warnings) and serves as a general indicator of how well the classifier performed. As mentioned before, our model offered an
accuracy of 0.805 over the test set. We must recall that we cannot directly compare this accuracy metric with the accuracy of any of the tools we used: The definitions of what constitutes a true or false negative for the tools are not the same as those for the classifier. For the tools, a false negative involves sites in Juliet for which they did not trigger any warnings, and our model does not consider such sites. Still, we can compare this value with those of related works. For instance, Ruthruff et al., 2008 could identify false positive warnings with 0.85 accuracy. We consider 0.805 to be an excellent result given the advantages we gained by not using information extracted from the analyzed source code other than the static analyzers warnings themselves, as already discussed.

**Precision (0.678)** is the number of True Positives divided by the sum of True and False Positives and indicates how rare are the False Positives. The result obtained from our model over the test set was rather low when compared to the values for recall and accuracy, at 0.678. This low value may be influenced by the unbalanced data, where 60.7% of the examples are on the false positive class. The unbalanced data, in turn, is due to the low precision of the static analysis tools themselves, which is 0.392 for the combined report\(^1\), as inferred from the aggregated tools data in Table 4.3. Figure 5.1 compares the precision of each individual tool and the aggregate of all tools with the precision of our predictive model, demonstrating that it provides considerable improvements over them.

**Recall (0.958)** is calculated by dividing the number of True Positives by the total number of Positive warnings in the input and indicates how many Positive warnings in the input succeed in being recognized as such. In contrast to Precision, the classification recall obtained for the test set was considerably high: 0.958. Since we achieved such high recall value, if we merely pruned all warnings classified as false positives by the model, the loss of information (true bugs discarded) would be less than 5%, while 70.6% of the noise (false alarms generated by the tools) would be discarded.

**F-score (0.794)** is the harmonic mean of Precision and Recall and serves to present a compound view of these two other metrics, i.e., to suggest how well the classifier performs in terms of keeping the False Positive rate low while not discarding much of the True Positives from the input. As we have already seen, we achieved high Recall and medium Precision rates; the F-score value of 0.794 showcases the good performance of the model in balancing these two goals.

Another aspect of interest of these results is the relevance of the features. As discussed in Section 2.1, the most relevant features for False Positive detection are derived from the source code itself. Since we avoid collecting such data, which other features are the most important for classification? Figure 5.2 shows that the most relevant characteristics for classification are the total number of warnings triggered for the file whose current warning was triggered for, the name of the tool that triggered the warning and the confirmations of each static analyzer for that warning. It is worth emphasizing that the feature referring

| Model Prediction | Warning | Labels |
|------------------|---------|--------|
|                  | TP      | FP     | Total |
| TP               | 2440    | 1158   | 3598  |
| FP               | 107     | 2784   | 2891  |
| **Total**        | 2547    | 3942   | 6489  |

Table 5.2: Confusion matrix

\(^1\)This is not the overall precision for the tools, but for the subset of warnings we could label according to Juliet documentation.
to the tool name is not independent of the three boolean features related to the tools confirmations. For instance, if Cppcheck appears as the tool name for a warning, the boolean feature for Cppcheck will be set to true, and the other two features, related to Frama-C and Clang Static Analyzer, will be true if the corresponding tool also triggers a warning for that same location and false if the corresponding tool does not trigger a warning for that same location. The redundancy level feature intent is to also capture situations where the same tool raises more than one warning for a given location.

5.2 Ranking Warnings

As mentioned before, we do not need to limit ourselves to a direct binary classification; it is also possible to use the trained predictive model to rank warnings according to their expected relevance. Next, we present and discuss the results obtained with our ranking approach and compare it with a random ranking algorithm. We compare our model to a random ranking algorithm instead of one of the models studied and presented as related works due to the complications related to experimental replication. For that, we would need to change original works to use our same tools and datasets. Furthermore, related works also used random ranking algorithms as a comparison base, thus, doing the same for our research allows comparisons with similar studies.

To evaluate our ranking performance over the test set, we refer to the methodology presented by Kremenek, Ashcraft, et al., 2004, which we describe below.

Figure 5.1: Precision of each individual tool and aggregated tools (precision of all tools combined), compared with our predictive model results.
Figure 5.2: AdaBoost classifier features importance: sum of the feature importances of each individual decision tree, divided by the total number of decision trees

1. We define $S(R)$ to be the sum of $FP_j$, the cumulative number of false positive warnings found before reaching the $j$th true positive warning when navigating a ranked list (starting from the first entry) ordered by a ranking algorithm $R$.

$$S(R) = \sum_{j=1}^{N_{tp}} FP_j$$  \hspace{1cm} (5.1)

It is worth observing that $S(R) = 0$ for an optimal ranking algorithm and $S(R) = N_{tp} \times N_{fp}$ for the worst ranking algorithm, where $N_{tp}$ and $N_{fp}$ are the total number of true positive warnings and false positive warnings in the list, respectively.

2. We then define the average of the cumulative number of false positive warnings found before reaching each true positive warning, $FP_{avg}$, as shown in Eq. (5.2).

$$FP_{avg} = \frac{S(R)}{N_{tp}}$$  \hspace{1cm} (5.2)

3. Finally, we measure the performance ratio of our ranking algorithm against a random ranking algorithm with Eq. (5.3).

$$Performance = \frac{FP_{avg}(random)}{FP_{avg}(AdaBoost \ ranking)}$$  \hspace{1cm} (5.3)
The random ranking algorithm randomly shuffles the test set. To avoid bias in our experiment, we run the random algorithm ten times, calculate the $FP_{\text{avg}}$ for each of them and use the median value as $FP_{\text{avg}}(\text{random})$ to calculate the performance ratio in Eq. (5.3). Table 5.3 summarizes the average number of cumulative false positives hit before each true positive is found when navigating the list from top to bottom for different ranking approaches over the test set (see Eq. 5.2).

| Ranking Approach         | Average False Positives |
|--------------------------|-------------------------|
| Perfect ranking          | 0                       |
| Worst case ranking       | 3942                    |
| Random ranking           | 1992                    |
| Predictive model ranking | 380                     |

Table 5.3: Average number of cumulative false positives hit before each true positive

![Graph showing number of actual flaws hit upon top-down inspection for different ranking approaches.](image)

Figure 5.3: Number of actual flaws hit upon top-down inspection of a ranked list for different ranking approaches. *kiskadee* refers to the predictive model proposed in this study.

The median model performance over random was 5.2, which indicates that, on average, one hits 5.2 times more false positives before each true positive with a random ranked warning list than one would if using the proposed ranking. Figure 5.3 shows the number of real flaws found in a ranked list per inspected entries (warnings) for different ranking models applied to the test set:

- *optimal*, where all the real flaws are at the top of the list;
• worst, where all the false positives are at the top of the list;
• random, where the entries are randomly shuffled in a list; and
• model, which represents the ranking model proposed in this paper while using the
classifier obtained during our experiments, as described above.

As Figure 5.3 shows, our model outperforms the random ranking algorithm by presenting
all software flaws in the test set after 3990 inspections, while the random ranking algorithm
presents software flaws in a linear relation with the number of inspections, where the last
few real software flaws in the test set are only presented at the end of the list, after 6486
inspections.

5.3 Discussion and Limitations

As can be observed by verifying the differences between the data in Table 4.2 and
Table 4.3, we only used 9% of the warnings generated by the static analysis tools over Juliet
inspections to compose our data set. We discarded the other 91% of the warnings generated
because they were not related to the injected bugs being tested for a given test case under
analysis: they were either unintentional flaws or false positives, and according to Juliet
documentation itself, one should not make assumptions about these warnings.

Despite this low proportion, the absolute number of warnings (25,959) was quite large
when compared to the sizes of data sets of similar studies. For instance, Ruthruff et al.,
2008 used a data set of 1,652 warnings in their study to predict actionable warnings. Our
data set size highlights the benefits of automatic labeling static analyzer warnings, which is
only feasible with synthetic test cases.

Although we used $T = 100$, according to the results of the cross-validation, it is
important to point out that we could not observe great variations for the different values of
$T$ we tested. Figure 5.4 shows how the area under the Receiver Operating Characteristic
(ROC) curve (used to compare binary classifiers) varies as we increase the number of trees
used in our model ($T$). As can be seen, the value converges very quickly, for values of $T$
as low as 5, and remains approximately constant up to $T = 150$, which was the highest value
tested during our experiments. At the same time, we did not observe the expected reduction
in the error rate as described by Freund et al., 1999. These two discrepancies in relation to
the expected behavior of the weak classifiers aggregate could be due to the imbalance of
examples present in our data set, as can be observed in Table 4.3. This imbalance comes
from the performance of the selected static analyzers used and might have inserted some
bias in the model training experiments. Another possible cause might be the difficulty in
classifying some examples for some specific cases. In any case, they do not invalidate our
results.

While the Accuracy, Precision, Recall, and F-Score obtained are interesting in themselves,
further analysis to compare the accuracy of the proposed model to those of the static analyzers
could be beneficial. We were not able to do so because we did not collect any data regarding
the false negatives generated by the tools.

Regarding the data set used for the experiments, we relied on Juliet 1.2 documentation,
provided by the US National Institute of Standards and Technology. Thus, we performed
little or no manual inspection on the warnings triggered by the tools, meaning that any
issues with the test suite or its documentation would also lead to mislabeled examples in
the data set. As already mentioned, due to limitations of the test suite for our purposes, we
also had to discard about 91% of the data generated by the static analyzers. This may have
influenced the tools false positive rates, which could interfere with our model predictions for certain flaw categories. Still regarding the dataset, it is important to point that the methodology presented in this work was applied to a single dataset (Juliet), which was developed with a certain purpose in mind, and that its examples and flaws may not be a good representation of real-world software. In this case, the model trained during our experiments would not be fit to arbitrate about warnings generated for other computer programs.

**Figure 5.4:** Area under ROC curve versus the number of trees used for AdaBoost
Chapter 6

Conclusions and Future Work

In this work, we developed and described kiskadee, a continuous source code static analysis tool that can monitor different software repositories and run multiple static analyzers on different software package versions. To rank the static analysis warnings in each report, based on the likelihood of a warning being a true positive, we presented a novel approach in which we train a prediction model to classify static analysis warnings from different static analyzers. Our model achieved 0.805 accuracy, 0.678 precision, and 0.958 recall when classifying our test data set.

Different from related works, kiskadee’s ranking approach does not use features based on the analyzed project intrinsic properties for model training, namely, source code change history and code metrics. Consequently, by smoothly decreasing the classification accuracy, the model obtained can be used successfully with any given software project. This is a compelling trade-off to build generic tools, like kiskadee, that can perform static analysis in any project provided as input and arbitrate on the positiveness of the alarms generated in this fashion, allowing the continuous monitoring and analysis of different software repositories, such as the ones provided by GNU/Linux distributions.

To answer our first research question (RQ1) of whether it is possible to rank or classify warnings using a dataset composed by the warnings only, i.e., without using the analyzed source code as input, we trained a predictive model capable of classifying multiple static analyzers warnings into true or false positives for any given software project (the features used in our ranking approach include the number of warnings triggered for a file and the name of the tools that triggered the warning). We then used our classifier to rank the warnings from these multiple static analyzers into a single report in a universal reporting notation. The ranking was organized by sorting the warnings in the report according to the model prediction probabilities: the ones with a higher likelihood of being actual software flaws are placed at the top of the list, and the warnings more likely to be false positives are placed at the bottom of the list. We showed how our approach compares to randomly sorting the warnings in the report: results show our ranking method performs over five times better than random ranking.

The methodology and tools used to build the data set of labeled static analysis warnings answer our second research question (RQ2) of whether it is possible to automate the labeling process. We were able to label 25959 warnings without individually inspecting them. The only manual step taken during the labeling process was comparing the tools warning messages with the categories of software flaws present in Juliet to comply with the test suite documentation and verify if a given warning was referring to the category of flaws being tested at a given moment.

For full reproducibility of our ranking experiments, the data sets and files used in
this study, including the alarms generated by examining the Juliet test suite with the different static analyzers used in this thesis, both in their original forms and converted to the Firehose universal report format, are publicly available at https://gitlab.com/ccsl-usp/kiskadee-ranking-data. These resources can also aid the reader to better understand the input and output used in each step described by our methodology.

While our ranking model does not offer the same performance as the related works with the best numerical results for ranking or classifying static analysis warnings, it comes close while offering a much more extensive horizon of possible applications since we do not use project specific features to rank the warnings. This means that by using our approach, an application does not need to have any previous knowledge about the software projects that were analyzed to generate the static analysis reports given as input to our model.

Supported by the ranking approach we developed, kiskadee can be used to reduce the cost of inspecting false alarms by setting a minimum value for the rate in which real flaws are found per inspection in a ranked list (i.e., a confidence level). When the rate of real flaws per inspections drops below that level, one could stop the inspection for that warning list. A variety of other tools to aggregate and display warnings from multiple static analyzers are both available in the industry and studied in the academia (Buckers et al., 2017; Heinemann et al., 2014). A promising application of the approach to rank static analyzer warnings presented in this thesis is to integrate these tools, which aggregate warnings from multiple static analyzers in dashboards, with the techniques introduced here.

kiskadee is licensed under the GNU Affero General Public License. Its development repository, the complete project documentation, and the data set used for the ranking experiments here presented are available at pagure.io/kiskadee.

6.1 Future Work

Juliet was the only dataset used to perform our experiments. It is important to find or build other datasets and include more static analyzers in the experiments to make the results presented during this thesis more solid. One possibility would be to manually build a first dataset by searching FLOSS repositories for patches that fix Common Vulnerabilities and Exposures (CVE) and tag that code section (before applying the patch) as buggy. Then, we could look for methods to automate this labeling steps and automate the construction of such dataset. This is left for future work.

It is possible to employ our ranking strategy to reduce the cost of inspecting false alarms by setting a minimum value for the rate in which real flaws are found per inspection in a ranked list (i.e., a confidence level), when the rate of real flaws per inspections drops below that level, one could stop the inspection for that warning list. Studying these confidence levels and the trade-off between loss of information and the cost of inspecting a more significant number of false alarms is left for future work.

Related works propose methods to identify bugs included or fixed in specific software versions, as mentioned in Section 2.1.2. Also, there are existing tools like csdiff that provide some logic to parse static analyzer reports from different software versions and list new and removed warnings. These tools and techniques could be used to filter warnings by version, so the ranked warning list displayed to the user would be a subset of the whole analysis report containing only the warnings introduced in a specific version of the analyzed software, reducing the scope of the manual inspections performed by software development teams. This could also aid in identifying false positives, as proposed by Kim and Ernst, 2007 since false positives tend to persist between software versions. As claimed by Spacco et al., 2006,

1https://github.com/kdudka/csdiff
tracking warnings across software versions is not as trivial as it may seem at first glance. Hence, integrating such techniques to kiskadee is left as future work.

Another interesting topic to be explored in future work is the study and design of a universal static analysis report standard. While Firehose meets our experimental needs both as a report format and as a tool with its parsers and generators, there is no evidence of research effort or validation on the report format design. Validating it or designing a new format validated by static analysis tool developers, practitioners and researchers would be a valuable contribution to the field.

Finally, during our experiments, we could observe the poor performance of some open source static analyzers. NIST have evaluated several source code static analyzers, both proprietary and open source, and published their results through the Static Analysis Tool Exposition (SATE) workshop reports (Okun et al., 2013). Their results present much lower false positive rates than the ones we found during our experiments. Although SATE reports do not provide the names of the evaluated tools when reporting data such as false positive rates, we believe that the proprietary tools outperform FLOSS tools on the field. Studying the static analysis techniques used by FLOSS tools and comparing them to the ones used by proprietary tools (when feasible) and to the state of the art is left as the last topic for future work.

6.2 Outlook

Source code static analysis can be reduced to the halting problem, hence, it is undecidable. This undecidability problem may drive researchers away from the field, since one may argue that there is no motivation to work on an issue known to be unsolvable. Recent studies have proven that applying different techniques to find heuristics to reduce the false positive rates of static analyzers provide means to produce tools that can find software flaws with false positive rates that are low enough to produce reports that are valuable to software developers and industry.

Studying new techniques to reduce the amount of false positives on the static analyzer level is a relevant field which still needs progress. Most of the static analysis tools that can produce reports with low false positive rates are proprietary tools and their internal techniques are not always publicly available. Open source static analyzer developers should be as close as possible from the academy so they can always test and implement new techniques in production level tools to reduce the false positive rate gaps that exist between proprietary and open source static analysis tools. Software engineering practitioners should also look into static analysis as a technique to improve code quality and prevent critical bugs to be released with production software versions.

Finally, open source communities should embrace static analysis, preferably with open source static analyzers, to both improve the quality of the code they ship and to provide feedback to static analyzer projects so they can keep improving their tools and techniques.
Appendix A

List of Papers Selected for Literature Review

This is a comprehensive list of papers selected to compose our work literature review as described in Chapter 2. Section A.1 lists the papers selected from our conference of choice (18th IEEE International Working Conference on Source Code Analysis and Manipulation - SCAM). Section A.2 lists the papers of interest extracted from SCAM papers that matched our search strings. Finally, Section A.2.1 is the subset of the papers in Section A.2 that were extracted from the *Survey of approaches for handling static analysis warnings* T. Muske and Serebrenik, 2016.

A.1 SCAM Selection

- A case study in detecting software security vulnerabilities using constraint optimization Weber et al., 2001
- Evaluating clone detection tools for use during preventative maintenance Burd and Bailey, 2002
- Static Analysis for Computing Escapability and Mutability for Java Components Shi and Naumovich, 2005
- Bug Classification Using Program Slicing Metrics Pan et al., 2006
- Characterising, Explaining, and Exploiting the Approximate Nature of Static Analysis through Animation Binkley et al., 2006
- Prioritizing Software Inspection Results using Static Profiling Boogerd and Moonen, 2006
- Parfait - A Scalable Bug Checker for C Code Cifuentes, 2008
- The Evolution and Decay of Staticaly Detected Source Code Vulnerabilities Penta et al., 2008
- A Value Analysis for C Programs Canet et al., 2009
- How Good is Static Analysis at Finding Concurrency Bugs? Kester et al., 2010
- Why Source Code Analysis and Manipulation Will Always be Important Harman, 2010
- Counting Bugs is Harder Than You Think P. E. Black, 2011
- Cooperative Testing and Analysis: Human-Tool, Tool-Tool and Human-Human Cooperations to Get Work Done Xie, 2012
- PtrTracker: Pragmatic Pointer Analysis Huuck et al., 2013
- Review Efforts Reduction by Partitioning of Static Analysis Warnings T. B. Muske et al., 2013
- Improving Prioritization of Software Weaknesses using Security Models with AVUS Renatus et al., 2015
- Survey of approaches for handling static analysis alarms T. Muske and Serebrenik, 2016

A.2 References Extracted From SCAM Papers

- Undecidability of Static Analysis Landi, 1992
- Improving Security Using Extensible Lightweight Static Analysis Evans and Larochelle, 2002
- Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations Kremenek and Engler, 2003
- Empirical Evaluation of Defect Projection Models for Widely-deployed Production Software Systems Li et al., 2004
- Finding Bugs is Easy Hovemeyer and Pugh, 2004
- Static analysis tools as early indicators of pre-release defect density Nagappan and Ball, 2005
- When Do Changes Induce Fixes? Śliwerski et al., 2005
- Tracking Defect Warnings Across Versions Spacco et al., 2006
- Which Warnings Should I Fix First? Kim and Ernst, 2007
- Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach Ruthruff et al., 2008
- Why Don’t Software Developers Use Static Analysis Tools to Find Bugs? Johnson et al., 2013

A.2.1 References Extracted From Survey

- Analyzing the state of static analysis: A large-scale evaluation in open source software Beller et al., 2016
- Correlation exploitation in error ranking Kremenek, Ashcraft, et al., 2004
- Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis Jung et al., 2005
• ISA: A source code static vulnerability detection system based on data fusion Kong et al., 2007
• An approach to merge results of multiple static analysis tools Meng et al., 2008
• Automated classification of static code analysis alerts: A case study Yüksel and Sözer, 2013
• Reducing false alarms from an industrial-strength static analyzer by SVM Yoon et al., 2014
• A model building process for identifying actionable static analysis alerts S. Heckman and Williams, 2009
Appendix B

Firehose XML schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
Copyright 2013 David Malcolm <dmalcolm@redhat.com>
Copyright 2013 Red Hat, Inc.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
02110-1301
-->
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <!-- Results from the invocation of an analysis tool -->
    <element name="analysis">
      <ref name="metadata-element"/>
      <element name="results">
        <zeroOrMore>
          <choice>
            <ref name="issue-element"/>
            <ref name="failure-element"/>
            <ref name="info-element"/>
          </choice>
        </zeroOrMore>
      </element>
      <optional>
        <ref name="custom-fields-element"/>
      </optional>
    </element>
  </start>
  <define name="metadata-element">
    <element name="metadata">
```

<element name="generator">
  <attribute name="name"/>
  <optional>
    <attribute name="version"/>
  </optional>
</element>

<!-- "sut" = "Software Under Test" -->
<optional>
  <element name="sut">
    <choice>
      <element name="source-rpm">
        <attribute name="name"/>
        <attribute name="version"/>
        <attribute name="release"/>
        <attribute name="build-arch"/>
      </element>
      <!-- Debian SUT entries -->
      <element name="debian-source">
        <!-- Report for a Debian source package -->
        <attribute name="name"/>
        <attribute name="version"/>
        <optional>
          <!-- This entry is optional because Debian packages may be 'native' (e.g. no local version, since local and upstream are the same). -->
          <attribute name="release"/>
        </optional>
        <!-- No build arch; source is arch indep. -->
      </element>
      <element name="debian-binary">
        <!-- Report for a Debian .deb package -->
        <attribute name="name"/>
        <attribute name="version"/>
        <optional>
          <attribute name="release"/>
        </optional>
        <attribute name="build-arch"/>
        <!-- Valid entries include 'amd64', 'kfreebsd-amd64', 'armhf', 'hurd-i386', among others for Debian. -->
      </element>
      <!-- What other options should we have? -->
    </choice>
  </element>
  <optional>
    <ref name="file-element"/>
  </optional>
  <optional>
    <element name="stats">
      <attribute name="wall-clock-time">
        <data type="float"/>
      </attribute>
    </element>
  </optional>
</optional>
<element>
</define>

<!-- Definitions of the various kinds of result follow:
<issue>
<failure>
<info>

--> A report about a possible problem -->
<define name="issue-element">
<element name="issue">
<optional>
<!-- The Common Weakness Enumeration ID
(see http://cwe.mitre.org/index.html )
e.g. "131" representing CWE-131
aka "Incorrect Calculation of Buffer Size"
http://cwe.mitre.org/data/definitions/131.html
-->
<attribute name="cwe">
<data type="integer"/>
</attribute>
</optional>

<optional>
<!-- Each static analysis tool potentially has multiple tests,
with its own IDs for its own tests.
Capture those that do here, as free-form strings:
-->
<attribute name="test-id"/>
</optional>

<optional>
<!-- Each static analysis tool potentially can report a "severity",
which may be of use for filtering.
The precise strings are likely to vary from tool to tool.
To avoid data-transfer issues, support storing it as an
optional
freeform string here.

See:
http://lists.fedoraproject.org/pipermail/firehose-devel
/2013-February/000001.html
-->
<attribute name="severity"/>
</optional>

<!-- A message summarizing the problem -->
<ref name="message-element"/>

<!-- Additional descriptive details
This might support some simple markup at some point
(as might <message>) -->
<optional>
<element name="notes"><text/></element>
</optional>
<!-- Where is the problem? -->
<ref name="location-element"/>

<optional>
  <!-- How can the problem occur? -->
  <element name="trace">
    <oneOrMore>
      <element name="state">
        <ref name="location-element"/>
        <optional>
          <element name="notes"><text/></element>
          </optional>
          </ref-- optionally we can supply key-value pairs -->
          <zeroOrMore>
            <element name="annotation">
              <element name="key"><text/></element>
              <element name="value"><text/></element>
            </element>
            </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </optional>
</define>

<!-- A given tool/testid may have additional key/value pairs that it
may be useful to capture: -->
<optional>
  <ref name="custom-fields-element"/>
</optional>
</define>

<!-- A report about a failed analysis.
If any of these are present then we don’t have full coverage.
For some analyzers this is an all-or-nothing affair: we either
get issues reported, or a failure happens (e.g. a segfault of the
analysis tool).
Other analyzers may be more fine-grained: able to report some
issues, but choke on some subset of the code under analysis. For example
cpychecker runs once per function, and any unhandled Python
exceptions only affect one function. -->
<define name="failure-element">
  <element name="failure">
    <optional>
      <!-- Each static analysis tool potentially can identify types of
way that it can fail.
Capture those that do here, as (optional) free-form strings:
Some analysis tools may be able to annotate a failure report by providing the location *within the software-under-test* that broke them.

For example, gcc-python-plugin has a gcc.set_location() method which can be used by a code analysis script to record what location is being analyzed, so that if unhandled Python exception happens, it is reported at that location. This is invaluable when debugging analysis failures.

Every type of failure seems to have its own kinds of data that are worth capturing:

* stdout/stderr/returncode for a failed subprocess
* traceback for an unhandled Python exception
* verbose extra information about a cppcheck failure etc.

Hence allow a <failure> to optionally contain extra key/value pairs, based on the failure-id.

Sometimes you may want a tool to report other kinds of information about the software-under-test that isn’t a problem as such, e.g. code metrics, copyright/license info, cross-referencing information etc, hence the <info> element:

An optional free-form string identifying the kind of information being reported:
<ref name="message-element"/>
</optional>
<optional>
<ref name="custom-fields-element"/>
</optional>
</element>
</define>
</element>

<!-- ...end of result definitions. Various supporting elements follow -->

<!-- Summary text aimed at a developer. This is required for an <issue>, but is also can (optionally) be provided by a <failure> or <info> -->
<define name="message-element">
<element name="message"><text/></element>
</define>

<define name="location-element">
<!-- A particular source code location -->
<element name="location">
<ref name="file-element"/>
</element>

<!-- Ideally, every analyzer would tell us in which function each problem was discovered, given that function names are less likely to change than line numbers. 

Unfortunately many don’t - and we should patch these in each upstream analyzer as we go. 

Also, a problem can occur in global scope (e.g. lack of NULL termination in an array-initializer for a global, such as in this checker: http://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html#verification-of-pymethoddef-tables (although arguably there *is* a relevant function there: the location of the code that uses that data) -->

<optional>
<element name="function">
<attribute name="name"/>
</element>
</optional>

<!-- We can refer to either a location, or a range of locations within the file: -->
<choice>
<ref name="point-element"/>
<element name="range">
<!-- start of range: -->
<ref name="point-element"/>
<!-- end of range: -->
<ref name="point-element"/>
</element>
</choice>
</element>
</define>
<define name="file-element">
  <element name="file">
    <!--
    What filename was given by the analyzer?
    This is typically the one supplied to it on the command line,
    which might be absolute or relative.
    Examples:
    - "foo.c"
    - "./src/foo.c"
    - "/home/david/libfoo-1.0/src/foo.c"
    -->
    <attribute name="given-path"/>
    <!--
    Optionally, record the absolute path of the file,
    to help deal with collating results from a build that
    changes working directory (e.g. recursive make)
    -->
    <optional>
      <attribute name="absolute-path"/>
    </optional>
    <optional>
      <element name="hash">
        <attribute name="alg"/>
        <attribute name="hexdigest"/>
      </element>
    </optional>
  </element>
</define>

<define name="point-element">
  <element name="point">
    <attribute name="line"/>
    <attribute name="column"/>
  </element>
</define>

<define name="custom-fields-element">
  <element name="custom-fields">
    <zeroOrMore>
      <choice>
        <element name="str-field">
          <attribute name="name"/>
        </element>
        <element name="int-field">
          <attribute name="name"/>
          <data type="integer"/>
        </element>
      </choice>
    </zeroOrMore>
  </element>
</define>
</grammar>
Bibliography

[CVE - Common Vulnerabilities and Exposures (CVE) n.d.] CVE - Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/. Last accessed on 2017-01-18 (cit. on p. 7).

[CWE - Common Weakness Enumeration n.d.] CWE - Common Weakness Enumeration. http://cwe.mitre.org/. Last accessed on 2017-01-18 (cit. on p. 7).

[Burd and Bailey 2002] Elizabeth Burd and John Bailey. “Evaluating clone detection tools for use during preventative maintenance”. In: Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on. IEEE. 2002, pp. 36–43 (cit. on p. 43).

[Boland and P. E. Black 2012] T. Boland and P. E. Black. “Juliet 1.1 C/C++ and Java Test Suite”. In: Computer 45.10 (Oct. 2012), pp. 88–90. issn: 0018-9162. doi: 10.1109/MC.2012.345 (cit. on p. 22).

[Beller et al. 2016] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. “Analyzing the state of static analysis: A large-scale evaluation in open source software”. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on. Vol. 1. IEEE. 2016, pp. 470–481 (cit. on p. 44).

[Binkley et al. 2006] David Binkley, Mark Harman, and Jens Krinke. “Characterising, explaining, and exploiting the approximate nature of static analysis through animation”. In: Source Code Analysis and Manipulation, 2006. SCAM’06. Sixth IEEE International Workshop on. IEEE. 2006, pp. 43–52 (cit. on p. 43).

[Paul E Black 2009] Paul E Black. “Static analyzers in software engineering”. In: The Journal of Defense Software Engineering 22.3 (2009), pp. 16–17 (cit. on pp. 1, 2).

[P. E. Black 2011] P. E. Black. “Counting Bugs is Harder Than You Think”. In: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. Sept. 2011, pp. 1–9. doi: 10.1109/SCAM.2011.24 (cit. on pp. 7, 44).

[Booger and Moonen 2006] Cathal Booger and Leon Moonen. “Prioritizing software inspection results using static profiling”. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. IEEE. 2006, pp. 149–160 (cit. on pp. 6, 8, 43).

[Paul E. Black and Ribeiro 2016] Paul E. Black and Athos Ribeiro. SATE V Ockham Sound Analysis Criteria. Tech. rep. NIST IR 8113. Data available
at http://dx.doi.org/10.18434/T4WC7V. National Institute of Standards and Technology, Mar. 2016. url: http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8113.pdf (visited on 01/16/2017) (cit. on pp. 7, 14, 15).

[Buckers et al. 2017] Tim Buckers et al. “UAV: Warnings from multiple Automated Static Analysis Tools at a glance”. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE. 2017, pp. 472–476 (cit. on p. 40).

[Canet et al. 2009] Géraud Canet, Pascal Cuoq, and Benjamin Monate. “A value analysis for C programs”. In: Source Code Analysis and Manipulation, 2009. SCAM’09. Ninth IEEE International Working Conference on. IEEE. 2009, pp. 123–124 (cit. on p. 43).

[Cifuentes 2008] Cristina Cifuentes. “Parfait-A scalable bug checker for C code”. In: Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on. IEEE. 2008, pp. 263–264 (cit. on p. 43).

[Drucker and Cortes 1996] Harris Drucker and Corinna Cortes. “Boosting decision trees”. In: Advances in neural information processing systems. 1996, pp. 479–485 (cit. on p. 28).

[Evans and Larochelle 2002] D. Evans and D. Larochelle. “Improving security using extensible lightweight static analysis”. In: IEEE Software 19.1 (Jan. 2002), pp. 42–51. issn: 0740-7459. doi: 10.1109/52.976940 (cit. on pp. 6, 44).

[Freund et al. 1999] Yoav Freund, Robert Schapire, and Naoki Abe. “A short introduction to boosting”. In: Journal-Japanese Society For Artificial Intelligence 14.771-780 (1999), p. 1612 (cit. on pp. 2, 28, 36).

[Harman 2010] Mark Harman. “Why source code analysis and manipulation will always be important”. In: Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on. IEEE. 2010, pp. 7–19 (cit. on p. 43).

[S. S. Heckman 2007] Sarah Smith Heckman. “Adaptively ranking alerts generated from automated static analysis”. In: Crossroads 14.1 (2007), p. 7 (cit. on p. 1).

[Heinemann et al. 2014] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. “Teamscale: Software quality control in real-time”. In: Companion Proceedings of the 36th International Conference on Software Engineering. ACM. 2014, pp. 592–595 (cit. on p. 40).

[Hovemeyer and Pugh 2004] David Hovemeyer and William Pugh. “Finding Bugs is Easy”. In: SIGPLAN Not. 39.12 (Dec. 2004), pp. 92–106. issn: 0362-1340. doi: 10.1145/1052883.1052895. url: http://doi.acm.org/10.1145/1052883.1052895 (visited on 01/17/2017) (cit. on pp. 1, 44).

[Huuck et al. 2013] Ralf Huuck, Franck Cassez, Mads Olesen, and Sebastian Biallas. “PtrTracker: Pragmatic pointer analysis”. In: (2013) (cit. on p. 44).

[S. Heckman and Williams 2009] Sarah Heckman and Laurie Williams. “A model building process for identifying actionable static analysis alerts”. In: Software Testing
Verification and Validation, 2009. ICST’09. International Conference on. IEEE. 2009, pp. 161–170 (cit. on pp. 2, 7, 26, 45).

[INSTRUCTION 2003] CNSS INSTRUCTION. “4009,“National Information Assurance Glossary,” Committee on National Security Systems, May 2003”. In: Formerly NSTISSI 4009 (2003) (cit. on p. 7).

[JOHNSON et al. 2013] B. JOHNSON, Y. SONG, E. MURPHY-HILL, and R. BOWDIDGE. “Why don’t software developers use static analysis tools to find bugs?” In: 2013 35th International Conference on Software Engineering (ICSE). May 2013, pp. 672–681. doi: 10.1109/ICSE.2013.6606613 (cit. on pp. 1, 2, 6, 44).

[JUNG et al. 2005] Yungbum JUNG, Jaehwang KIM, Jaeho SHIN, and Kwangkeun Yi. “Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis”. In: International Static Analysis Symposium. Springer. 2005, pp. 203–217 (cit. on pp. 2, 3, 7, 21, 26, 44).

[KREME NEK and ENGLER 2003] Ted KREME NEK and Dawson ENGLER. “Z-ranking: Using statistical analysis to counter the impact of static analysis approximations”. In: International Static Analysis Symposium. Springer. 2003, pp. 295–315 (cit. on pp. 1, 6, 8, 44).

[KIM and ERNST 2007] Sunghun KIM and Michael 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. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 45–54. isbn: 978-1-59593-811-4. doi: 10.1145/1287624.1287633. url: http://doi.acm.org/10.1145/1287624.1287633 (visited on 01/17/2017) (cit. on pp. 6, 8, 40, 44).

[KESTER et al. 2010] Devin KESTER, Martin MWEBESA, and Jeremy S BRADBURY. “How good is static analysis at finding concurrency bugs?” In: Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on. IEEE. 2010, pp. 115–124 (cit. on p. 43).

[KONG et al. 2007] Deguang KONG, Quan ZHENG, Chao CHEN, Jianmei SHUAI, and Ming ZHU. “ISA: a source code static vulnerability detection system based on data fusion”. In: Proceedings of the 2nd international conference on Scalable information systems. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2007, p. 55 (cit. on p. 45).

[KREME NEK, ASHCRAFT, et al. 2004] Ted KREME NEK, Ken ASHCRAFT, Junfeng YANG, and Dawson ENGLER. “Correlation exploitation in error ranking”. In: ACM SIGSOFT Software Engineering Notes. Vol. 29. 6. ACM. 2004, pp. 83–93 (cit. on pp. 1, 2, 7, 21, 26, 33, 44).

[LANDI 1992] William LANDI. “Undecidability of static analysis”. In: ACM Letters on Programming Languages and Systems (LOPLAS) 1.4 (Dec. 1992), pp. 323–337. issn: 1057-4514. DOI: 10.1145/161494.161501. url: http://dl.acm.org/citation.cfm?id=161494.161501 (visited on 01/05/2017) (cit. on pp. 1, 44).
[Li et al. 2004] Paul Luo Li, Mary Shaw, Jim Herbsleb, Bonnie Ray, and Peter Santhanam. “Empirical evaluation of defect projection models for widely-deployed production software systems”. In: ACM SIGSOFT Software Engineering Notes. Vol. 29. 6. ACM. 2004, pp. 263–272 (cit. on p. 44).

[T. B. Muske et al. 2013] Tukaram B Muske, Ankit Baid, and Tushar Sanas. “Review efforts reduction by partitioning of static analysis warnings”. In: Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on. IEEE. 2013, pp. 106–115 (cit. on pp. 1, 6, 8, 44).

[Meng et al. 2008] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. “An approach to merge results of multiple static analysis tools (short paper)”. In: Quality Software, 2008. QSIC’08. The Eighth International Conference on. IEEE. 2008, pp. 169–174 (cit. on p. 45).

[T. Muske and Serебреник 2016] Tukaram Muske and Alexander Serебреник. “Survey of approaches for handling static analysis alarms”. In: Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. IEEE. 2016, pp. 157–166 (cit. on pp. 6, 43, 44).

[NASA 2004] NASA NASA. “Software Assurance Standard”. In: NASA Technical Standard NASA-STD-8739.8, National Aeronautics and Space Administration (NASA), http://standards.nasa.gov (July 2004) (cit. on p. 7).

[Nagappan and Ball 2005] Nachiappan Nagappan and Thomas Ball. “Static analysis tools as early indicators of pre-release defect density”. In: Proceedings of the 27th international conference on Software engineering. ACM. 2005, pp. 580–586 (cit. on p. 44).

[Okun et al. 2013] Vadim Okun, Aurelien Delaittre, and Paul E Black. “Report on the static analysis tool exposition (sate) iv”. In: NIST Special Publication 500 (2013), p. 297 (cit. on p. 41).

[Penta et al. 2008] M. D. Penta, L. Cerulo, and L. Aversano. “The Evolution and Decay of Statically Detected Source Code Vulnerabilities”. In: 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. Sept. 2008, pp. 101–110. doi: 10.1109/SCAM.2008.20 (cit. on pp. 6, 8, 43).

[Pan et al. 2006] K. Pan, S. Kim, and E. J. Whitehead Jr. “Bug Classification Using Program Slicing Metrics”. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. Sept. 2006, pp. 31–42. doi: 10.1109/SCAM.2006.6 (cit. on pp. 2, 43).

[Polikar 2006] Robi Polikar. “Ensemble based systems in decision making”. In: IEEE Circuits and systems magazine 6.3 (2006), pp. 21–45 (cit. on p. 28).

[Renatus et al. 2015] Stephan Renatus, Corrie Bartelheimer, and Jörn Eichler. “Improving prioritization of software weaknesses using security models with AVUS”. In: Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on. IEEE. 2015, pp. 259–264 (cit. on p. 44).
[Red Hat n.d.] Inc. Red Hat. Anitya - release-monitoring. https://release-monitoring.org/. Último acesso em 22/5/2018 (cit. on p. 13).

[Russell and Norvig 2003] Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. 2nd ed. Pearson Education, 2003. ISBN: 0137903952 (cit. on pp. 3, 9, 27).

[Ruthruff et al. 2008] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. “Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach”. In: Proceedings of the 30th International Conference on Software Engineering. ICSE ’08. New York, NY, USA: ACM, 2008, pp. 341–350. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368135. URL: http://doi.acm.org/10.1145/1368088.1368135 (visited on 01/18/2017) (cit. on pp. 2, 6–8, 21, 26, 32, 36, 44).

[Spacco et al. 2006] Jaime Spacco, David Hovemeyer, and William Pugh. “Tracking Defect Warnings Across Versions”. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. MSR ’06. New York, NY, USA: ACM, 2006, pp. 133–136. ISBN: 978-1-59593-397-3. DOI: 10.1145/1137983.1138014. URL: http://doi.acm.org/10.1145/1137983.1138014 (visited on 01/05/2017) (cit. on pp. 6, 8, 40, 44).

[Shi and Naumovich 2005] Aiwu Shi and Gleb Naumovich. “Static analysis for computing escapability and mutability for Java components”. In: Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on. IEEE. 2005, pp. 119–128 (cit. on p. 43).

[Sliwerski et al. 2005] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When do changes induce fixes?” In: ACM sigsoft software engineering notes. Vol. 30. 4. ACM. 2005, pp. 1–5 (cit. on p. 44).

[Weber et al. 2001] Michael Weber, Viren Shah, and Chris Ren. “A case study in detecting software security vulnerabilities using constraint optimization”. In: Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on. IEEE. 2001, pp. 1–11 (cit. on p. 43).

[Xie 2012] Tao Xie. “Cooperative testing and analysis: Human-tool, tool-tool and human-human cooperations to get work done”. In: Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on. IEEE. 2012, pp. 1–3 (cit. on p. 44).

[Xypolytos et al. 2017] Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr MT Ali-Eldin. “A Framework for Combining and Ranking Static Analysis Tool Findings Based on Tool Performance Statistics”. In: Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on. IEEE. 2017, pp. 595–596 (cit. on p. 9).

[Yoon et al. 2014] Jongwon Yoon, Minsik Jin, and Yungbum Jung. “Reducing false alarms from an industrial-strength static analyzer by SVM”. In: Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. Vol. 2. IEEE. 2014, pp. 3–6 (cit. on pp. 3, 7, 21, 45).
[YÜKSEL and SÖZER 2013] U. YÜKSEL and H. SÖZER. “Automated Classification of Static Code Analysis Alerts: A Case Study”. In: 2013 IEEE International Conference on Software Maintenance. Sept. 2013, pp. 532–535. doi: 10.1109/ICSM.2013.89 (cit. on pp. 21, 45).