Ranking Warnings from Multiple Source Code Static Analyzers via Ensemble Learning

Athos Ribeiro
FLOSS Competence Center
University of São Paulo, Brazil
athoscr@ime.usp.br

Paulo Meirelles
Federal University of
São Paulo, Brazil
paulo.meirelles@unifesp.br

Nelson Lago, Fabio Kon
FLOSS Competence Center
University of São Paulo, Brazil
{lago,kon}@ime.usp.br

ABSTRACT
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 looking for bugs in the source code. Combining the analysis of different tools may reduce the number of false negatives, but yields a corresponding increase in the absolute number of false positives (which is already high for many tools). A possible solution, then, is to filter these results to identify the issues least likely to be false positives. In this study, we post-analyze the reports generated by three tools on synthetic test cases provided by the US National Institute of Standards and Technology. In order 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 in the source code.

Author Keywords
Source code static analysis; Vulnerability warnings; Free Software; Open Source Software.

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. 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 [1]. 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 [17], 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 warning[1] that does not reveal a real flaw in the analyzed program, i.e., a false positive (or false alarm), or to fail generating a warning to a real flaw, i.e., a false negative.

Obviously, false negatives are undesirable, which might prompt the development of tools that try to identify as much flaws as possible. However, it is not hard to realize that attempts to minimize false negatives in a static analyzer are likely to increase the rate of false positives and vice-versa. Currently, empirical studies show that even carefully designed static analyzers may generate warnings at false positive rates that can exceed 30% [15]. Such false alarms require manual inspection, increasing the effort of analyzing tool reports when they include significant amounts of false positives [10, 19] and hindering the usage and general adoption of automated source code analyzers [12]. This means static analyzers today sit uncomfortably in a place where they let many flaws slip and, at the same time, impose a significant toll on the developer that has to sift through a vast amount of irrelevant warnings. A substantial improvement in this area could foster a wider 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 software.

The reduction of the false positive rates obtained from static analysis — and, consequently, of the amount of counter-productive information generated by static analyzers — can smooth the process of manually verifying analyzer reports, making them more useful and allowing tools to be less for-

OpenSym '19, August 20–22, 2019, Skövde, Sweden

ISBN 978-1-4503-6319-8/19/08...$15.00
DOI: https://doi.org/10.1145/3306446.3340828

[1]When 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.

giving of false negatives. This has prompted the elaboration of several studies that propose either pruning or ranking the warnings from static analysis reports to deal with false alarms [9, 13, 15, 22]. 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 hopefully serve as an entry point for enhanced static analysis reports without the need of meticulous pre-processing and training steps for each inspected program. 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 [1]. As we just discussed, using multiple tools also inflates the absolute number of false positives. This downside, however, is an acceptable compromise if paired with a corresponding improvement in the *rate* of false positives — the core of this work.

To experiment with this hypothesis, we created and trained a predictive model by processing a synthetic static analysis test suite, which is a collection of source code snippets with specific flaws injected in known locations. We ran multiple static analyzers using the test suite as input and converted the analysis reports to a universal format to ease further processing. Then, we checked whether every single warning matched one of the injected flaws in the test suite, labeling each either as true positive or false positive[2]. After the labeling step, we extracted the characteristics used to train our model. These did not include any characteristics from the analyzed source code and project history, which tend to be the most relevant ones when predicting warnings positiveness, as shown in previous works [13, 24]. To compensate for this, we used an ensemble learning method [21] to train several weak classifiers, which were then able to vote about the positiveness of new examples. Finally, instead of just classifying new examples as true and false positives, we ranked the warnings based on their probabilities of being real flaws, where the top entries were more likely to be of interest, and the last entries were more likely to be false positives.

We guided this investigation based on the following research questions:

RQ1: *Is it possible to rank or classify static analysis warnings without also manually inspecting and preprocessing the analyzed source code?* This would make such results more readily usable for a large number of projects.

RQ2: *Is it possible to partially or completely automate the process of labeling warnings as false positives?* This would enable studies with large data sets in environments where manually inspecting warnings is not a frequent activity.

The results obtained were very promising: using three static analyzers with an aggregate false positive rate of 0.61, we achieved an accuracy of 0.8, not too distant from the state of the art (0.85) which depends on direct source code analysis. During this work, we also generated a publicly available data set of static analysis alarms from multiple static analyzers on a universal report format, generated by analyzing a synthetic static analysis test suite.

[2]Observe that we do not discuss false negatives in this work.

The rest of this paper begins with 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). As we will see, this entails significant effort for every single program to be analyzed, highlighting the utility of avoiding this step. After that, we present our experience with the approach discussed. In Section 3, 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 and, in Section 4, 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 Section 5.

RELATED WORK

Muske and Serebrenik [18] 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. We assessed the studies and techniques under the aforementioned classification to better position the present work.

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 [9, 13, 15, 22, 24]. Although these project-specific features are in great part responsible for the high accuracy of the models proposed 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 [3] presents 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 [16] 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.

In a later work, Muske [19] points out the difficulties in reviewing a high number of warnings and divides the warnings into different categories to identify redundant warnings and estimate which categories are more prone to hold false positives. This work also mentions 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. [22] 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.

Kim et al. [14] mention the high rates of false positives in static analysis and claim that sometimes static analyzers prioritize their warnings inefficiently. They proceed to 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 show that the software change history may be important to prioritize warnings.

In another work, Penta et al. [5] perform an analysis of the evolution of vulnerabilities detected by static analyzers in three different open source applications: Samba, Squid, and Horde. The authors analyze how the number of vulnerabilities varies over time. Since they analyze different development versions (not only releases), they are able to include aspects of the development process, like bug fixing efforts right before a release. The work aims 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 (Section 1). This 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 [21].

Xypolytos et al. [23] propose 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 have not yet provide statistically significant results.

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. At the second level, we associate each warning to specific features that will compose a data set to train our prediction model.

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 [13, 15, 22, 24, 25]. 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 [22], 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 known, facilitating studies and freeing researchers from performing expensive searches along their data sets.

We use Juliet [2] version 1.2 to generate our data set. Juliet is a synthetic C/C++ test suite with 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 capture true positives) and an additional section with a correct, fixed instance of that previous flawed section (to capture 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 3 static analysis tools to generate the data set for our experiments. The criteria to select the tools were straightforward:

1. The tool must be able to examine C/C++ code for security flaws (e.g., buffer overflows, null pointer dereferences); and
2. the source code of the tool must be released under an open source license.

Criterion 1 ensures the tool can analyze a subset of Juliet test cases, 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)[3]. Open Source tools also simplify the process of retrieving string constructs for tool messages and categories when necessary, since we can verify their source code. Table 1 introduces the tools here employed to generate data through Juliet test cases analyses.

Tool	Target Language	License	Version
Clang	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 1: Selected static analysis tools

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. 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.

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, $\{J - (T_1 \cup T_2 \cup \ldots T_n)\}$ 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 $\{T_1 \cup T_2 \cup \ldots T_n\}$: test cases that cannot be addressed by some tool will not be in both cases. Again, focusing on negatives would invalidate this.

The only truly different approach, therefore, is to analyze only $\{T_1 \cap T_2 \cap \ldots T_n\}$. 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., 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 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.

[3] 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.

However, before examining Juliet with the selected static analysis tools, we still prune the test suite to limit it to software that can run on free OSes. Table 2 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 2: Number of Juliet test cases

Collecting Labeled Warnings From Multiple Tools

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 3 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 3: Total number of warnings generated per tool

Seeing that each tool has its own report format, we reduce the effort of parsing reports by converting them all to a unified report (UR) in a common format. 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 subsection. Table 4 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].

[4] Again, note we do not take false negatives into account in this work

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: Labeled warnings per tool

Extracting Features from Labeled Warnings

We extract 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 [9, 13, 15, 22]. For instance, Kremenek et al. [15] 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 are able to produce a more generic model, hopefully applicable to different contexts without the need to perform the expensive training step for each project one may want to analyze.

Our set of features is extracted by processing the aggregated report of labeled warnings. While the name of the tool that triggered a warning, the programming language analyzed, and the severity of the warning can be inferred 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.

With the exception of 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 5. 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 (each tool subdivides warnings in different numbers of categories, so we lumped the less critical ones below 3). Thus, If a tool does not provide severity information, we assume 4.

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 5: Possible Assigned Warning Categories

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;
- **Category:** Category of the warning, as shown in Table 5;
- **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.

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.

Training Decision Trees with AdaBoost

To make our model applicable to a wider 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., $\varepsilon < 0.5$ where ε is the classification error of the classifier. Ensemble learning methods combine the prediction of weak classifiers to build stronger classifiers with lower errors [21]. One widely used ensemble learning method is boosting [21], 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 this study, we chose to use the AdaBoost algorithm [8].

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 [8]. Each weak classifier is assigned a voting weight (α) based on its importance. The strong binary classifier (**H**) is given by Eq. (1), where **x** is a new example to be classified.

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{1}$$

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 3, 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 [6].

We divide our data set (the list of labeled features obtained in Section 3) into a training set and a test set. The training set is built, as usual, 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 10 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.

Ranking Static Analysis Warnings

Even though techniques to use AdaBoost on ranking problems are available in literature [7], we take a simpler approach and rank warnings based on AdaBoost 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 inspect only 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.

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.

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 a 10-fold cross-validation, as can be seen in Table 6.

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

Table 6: Mean training classification accuracy (cross-validation)

Accuracy is just one of the interesting metrics we can derive from the experiments. Table 7 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.

		True class		
		Real flaw	FP	Total
Classified as	Real flaw	2440	1158	3598
	FP	107	2784	2891
	Total	2547	3942	6489

Table 7: Confusion matrix. FP: False Positives or False alarms

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. [22] 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[5], as inferred from the aggregated tools data in Table 4. Figure 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, 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 2 shows that the most relevant characteristics for classification

[5]This is not the overall precision for the tools, but for the subset of warnings we could label according to Juliet documentation.

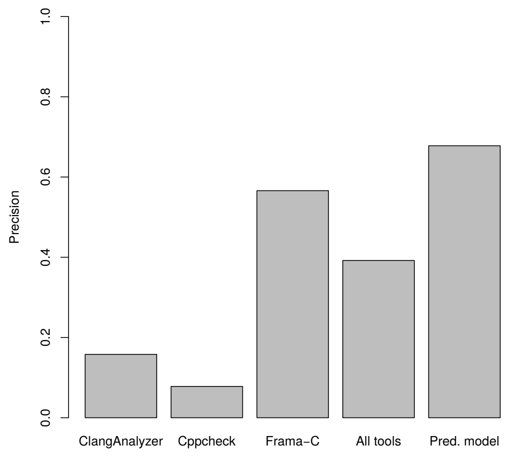

Figure 1: Precision of each individual tool and precision of all tools combined, compared with the predictive model results

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 to the tool name is not independent of the 3 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.

Ranking

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. To evaluate our ranking performance over the test set, we refer to the methodology presented by Kremenek et al. [15]:

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.

$$S(R) = \sum_{j=1}^{N_{tp}} FP_j \tag{2}$$

   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.

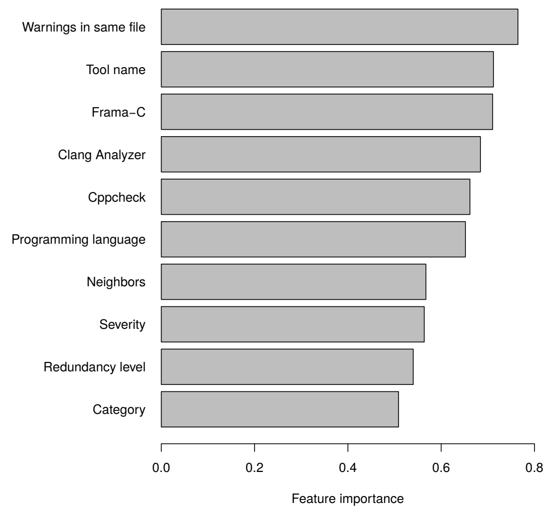

Figure 2: AdaBoost classifier features importance: sum of the feature importances of each individual decision tree, divided by the total number of decision trees

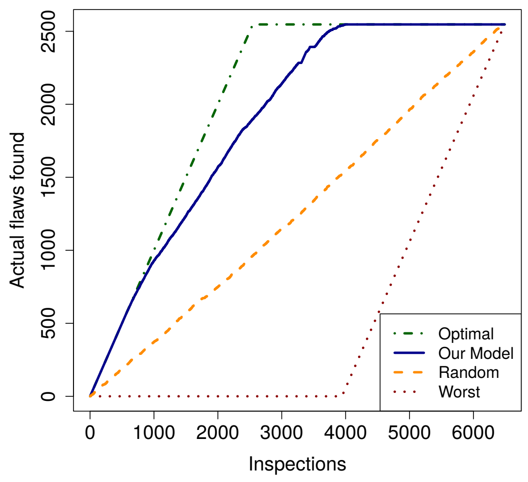

Figure 3: Number of actual flaws hit upon top-down inspection of a ranked list for different ranking approaches. **Our Model** refers to the predictive model proposed in this study

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. (3).

$$FP_{avg} = \frac{S(R)}{N_{tp}} \tag{3}$$

3. Finally, we measure the performance ratio of our ranking algorithm against a random ranking algorithm with Eq. (4).

$$Performance = \frac{FP_{avg}(random)}{FP_{avg}(\text{AdaBoost ranking})} \tag{4}$$

The random ranking algorithm randomly shuffles the test set. To avoid bias in our experiment, we run the random algorithm 10 times, calculate the FP_{avg} for each of them and use the median value as $FP_{avg}(random)$ to calculate the performance ratio in Eq. (4). Table 8 summarizes the average number of cumulative false positives hit before each true positive is found when navigating the list from the top to the bottom for different ranking approaches over the test set (see Eq. 3).

Perfect ranking	0
Worst case ranking	3942
Random ranking	1992
Predictive model ranking	380

Table 8: Average number of cumulative false positives hit before each true positive

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 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 in the top of the list;
- *worst*, where all the false positives are in 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 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 in the end of the list, after 6486 inspections.

Discussion and Limitations

As can be observed by verifying the differences between the data in Table 3 and Table 4, 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. Therefore, we do not know whether they are unintentional flaws on the tested code 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. [22] used a data set of 1,652 warnings on 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. We also should note that, although the performance of the tools we used varies a lot, removing any one of them would produce a significant impact in the results.

The actual results we achieved were not as good as the state of the art. We argue that this is a tradeoff, because the technique allows us to build a generic model that may be used with different software projects. We have not, however, tested this hypothesis with other code bases. Since the classifier is trained with Juliet, how well it does or does not perform with other code is directly related to how well Juliet covers the most significant software flaws. This also means the classifier we trained is strongly biased to search for security flaws (the core concern of Juliet), but the method should work with other kinds of flaws too, as long as an adequate training set is used.

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 reductions in the error rate as described in [8] for the different values of T we tested. This discrepancy 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. 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. Still, we should note that AdaBoost is prone to overfitting.

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.

CONCLUSIONS AND FUTURE WORK

In this paper, we described a novel approach in which we train a prediction model to classify static analysis warnings from different tools. Our model achieved 0.805 accuracy, 0.678 precision, and 0.958 recall when classifying our test data set. Our approach does not use features based on the analyzed project internal properties, namely, source code change history and code metrics. This means that, in contrast with related works that propose models with slightly better classification accuracies, our model might be used successfully with any given software project, which is an interesting trade-off to build generic tools to be used as entry points to automated source code static analysis.

To answer our first research question (*Is it possible to rank or classify static analysis warnings without also manually inspecting and preprocessing the analyzed source code?*), we trained a predictive model capable of classifying multiple static analyzers warnings into true or false positives for any given software project. 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 5 times better than random ranking. Future works should evaluate the performance of the model on multiple actual software projects.

The methodology and tools used to build the data set of labeled static analysis warnings answer our second research question (*Is it possible to partially or completely automate the process of labeling warnings as false positives?*). 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, 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 paper, both in their original forms and converted to a universal report format, are publicly available at `https://github.com/LSS-USP/kiskadee-ranking-data`.

As future work, it is possible to employ the ranking strategy presented in this paper 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 confidence levels and the trade-off between loss of information and the cost of inspecting a larger number of false alarms is left for future works.

As discussed, the current system ranks warnings more likely to correspond to actual flaws earlier. However, since different flaws have different practical impact, it might be useful to categorize flaws into a few groups (security vulnerabilities, UI errors, possible crashes etc.) and assign different weights to each one. This would allow for more relevant flaws with less certainty to be ranked before less relevant flaws with more certainty, which is probably more useful for the developer.

A variety of tools to aggregate and display warnings from multiple static analyzers are both available in the industry and studied in the academia [4, 11]. A promising application of the approach to rank static analyzer warnings presented in this paper is to integrate these tools (that aggregate warnings from multiple static analyzers in dashboards) with the techniques introduced here. In this context, we have been developing Kiskadee [20], a continuous static analysis tool that monitors software repositories for new releases and runs multiple static analyzers on each new version of the monitored software. Kiskadee aggregates the static analysis reports produced and employs the techniques presented in this paper to rank warnings.

Acknowledgments

This research was supported by CAPES (Coordination for the Improvement of Higher Education Personnel) and CNPq (National Council for Scientific and Technological Development), proc #311174/2017-5.

REFERENCES

1. Paul E Black. 2009. Static analyzers in software engineering. *The Journal of Defense Software Engineering* 22, 3 (2009), 16–17.
2. T. Boland and P. E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (Oct 2012), 88–90.
3. Cathal Boogerd and Leon Moonen. 2006. Prioritizing software inspection results using static profiling. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 149–160.
4. Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. 2017. 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, 472–476.
5. M. Di Penta, L. Cerulo, and L. Aversano. 2008. The Evolution and Decay of Statically Detected Source Code Vulnerabilities. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 101–110.
6. Harris Drucker and Corinna Cortes. 1996. Boosting decision trees. In *Advances in neural information processing systems*. 479–485.
7. Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. 2003. An efficient boosting algorithm for combining preferences. *Journal of machine learning research* 4, Nov (2003), 933–969.
8. Yoav Freund, Robert Schapire, and Naoki Abe. 1999. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence* 14, 771-780 (1999), 1612.
9. Sarah Heckman and Laurie Williams. 2009. A model building process for identifying actionable static analysis alerts. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 161–170.
10. Sarah Smith Heckman. 2007. Adaptively ranking alerts generated from automated static analysis. *Crossroads* 14, 1 (2007), 7.
11. Lars Heinemann, Benjamin Hummel, and Daniela Steidl. 2014. Teamscale: Software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 592–595.
12. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681.
13. Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. *Static Analysis* (2005), 203–217.
14. Sunghun Kim and Michael D. Ernst. 2007. 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)*. ACM, New York, NY, USA, 45–54.
15. Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 83–93.
16. Ted Kremenek and Dawson Engler. 2003. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*. Springer, 295–315.
17. William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (Dec. 1992), 323–337.
18. Tukaram Muske and Alexander Serebrenik. 2016. Survey of approaches for handling static analysis alarms. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE, 157–166.
19. Tukaram B Muske, Ankit Baid, and Tushar Sanas. 2013. Review efforts reduction by partitioning of static analysis warnings. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 106–115.
20. Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. 2018. Ranking Source Code Static Analysis Warnings for Continuous Monitoring of FLOSS Repositories. In *Open Source Systems: Enterprise Software and Solutions*, Ioannis Stamelos, Jesus M. Gonzalez-Barahoña, Iraklis Varlamis, and Dimosthenis Anagnostopoulos (Eds.). Springer International Publishing, Cham, 90–101.
21. Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.
22. Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 341–350.
23. Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr MT Ali-Eldin. 2017. 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, 595–596.
24. Jongwon Yoon, Minsik Jin, and Yungbum Jung. 2014. Reducing false alarms from an industrial-strength static analyzer by SVM. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, Vol. 2. IEEE, 3–6.
25. U. Yüksel and H. Sözer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *2013 IEEE International Conference on Software Maintenance*. 532–535.