Ranking Source Code Static Analysis Warnings for Continuous Monitoring of FLOSS Repositories

Athos Ribeiro, Paulo Meirelles, Nelson Lago, Fabio Kon

Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories

Athos Ribeiro[1], Paulo Meirelles[1,2], Nelson Lago[1], Fabio Kon[1]

[1] FLOSS Competence Center – University of São Paulo
{athoscr,lago,fabio.kon}@ime.usp.br
[2] Department of Health Informatics – Federal University of São Paulo
paulo@softwarelivre.org

Abstract. Performing source code static analysis during the software development cycle is a difficult task. There are different static analyzers available, and each of them usually works better in a small subset of problems, making it hard to choose a single tool. Combining the analysis of different tools solves this problem, but brings about other problems, namely the generated false positives and a large amount of unsorted alarms. This paper 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. 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: Static analysis, Software quality, False positives, Free Software, Open Source Software.

1 Introduction

Source code static analysis is a valuable technique to support software assurance. In theory, it can explore abstractions of all possible program behaviors, which is not feasible with software testing [11]. Thus, it can find software bugs using a complementary approach to automated tests.

However, the fundamental problems of static analysis are undecidable [15], so approximations must be made, leading static analyzers to generate false alarms or to miss occurrences of software flaws. *False positives* are produced when a static analyzer processes bug-free code and reports it as buggy code. The tool may also miss actual bugs when it processes buggy code and report it as a bug-free code [5] (*false negatives*). 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.

The high amount of information generated combined with a significant false alarm rate hinder the inclusion of static analyzers in the software development

cycle. Moreover, false positives require manual inspection, which increases the effort of analyzing tool reports [10, 17] and may even cause static analyzers to be discarded as irrelevant [14]. Literature suggests that using multiple static analyzers improves static analysis coverage [5] since some tools perform better in specific tasks due to different analysis methods. This practice may decrease the number of false negatives but is likely to generate more false positives as the number of tools used increases.

In this study, we present *kiskadee*, a system designed to support *continuous* static analysis in software repositories using multiple static analyzers to generate reports using a common output 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 [8] 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, providing them with more accurate data and favoring the use of static analysis.

2 Related Work

Muske and Serebrenik [16] provided an overview of different techniques on how to handle 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 [9,12,13,20,21]. Since 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.

Ruthruff et al. [20] propose a method 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 in the authors study, are the priority given by the static analyzer, the file length, and code indentation, suggesting that the authors performed additional code analysis to extract factors from the code.

As discussed, related works emphasize that the most important characteristics to arbitrate on static analysis warnings positiveness are internal to the

analyzed project. Our study differs from them by assessing static analysis warnings only with the information present in the warnings themselves, therefore, the approach in itself usually produces poor results. To compensate for this, we use multiple static analyzers with kiskadee to generate more information and correlate the information provided by them to assess the correctness of a given warning better. 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 [19].

3 Continuous Static Analysis with kiskadee

In Free/Libre and 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.

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.

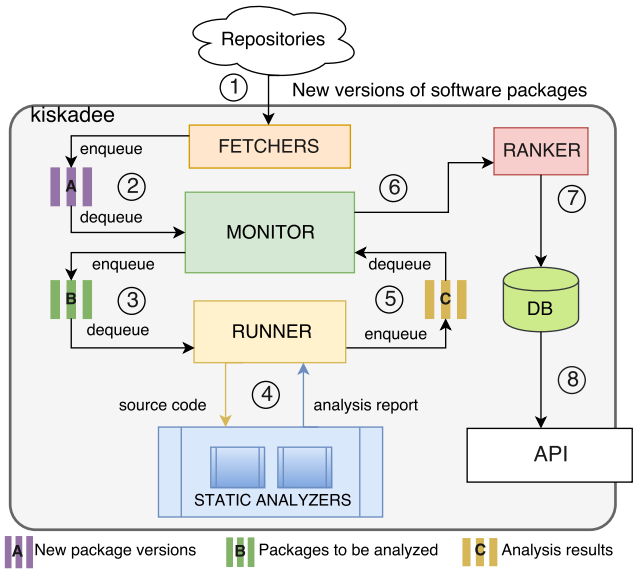

Fig. 1: kiskadee design overview.

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.

To continuously run static analysis on software packages and handle the false alarms generated by these analyses, we developed *kiskadee*. Figure 1 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. This ranking step is performed with a classification model, described in Section 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.

FLOSS development communities have been discussing a common static analysis report output format. The Fedora Project Static Analysis Special Interest Group [3] designed a tool to run static analyzers during the package build process [2]. Although the tool itself is in its early development stages and not ready for usage in production, the developers discussed [4] a common report format for the static analyzers in their mailing lists. After a few iterations, with Debian Project developers collaboration, they created *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. 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.

kiskadee was run with three static analysis tools to generate the warnings data set for our experiments. The Criteria to select the tools were: (1) The tool must be able to examine C/C++ code for security flaws (e.g., buffer overflows, null pointer dereferences); and (2) the tool source code must be released under an FLOSS license.

Criterion 1 ensures the tool can analyze a subset of the test cases in our data set, introduced in Section 4, 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).

Fedora Project uses an external system to monitor the software projects they distribute. Anitya [1] 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. Therefore, kiskadee monitors packages by reading information published by Anitya in the Fedora infrastructure publish-subscribe system.

kiskadee can point to 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.

4 Ranking Warnings

To create and train a predictive model, before analyzing project repositories, we ran kiskadee on Juliet [6], a publicly available test suite composed of a collection of source code snippets with specific flaws injected in known locations, which facilitates the assessment of static analysis tools. By running kiskadee on Juliet (without the ranking step), we obtained the analysis reports (a set of warnings) in the Firehose format, easing further processing. Then, we checked whether each single warning matched one of the injected flaws in the test suite, labeling them either as true positive or false positive. 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 [12,21]. By not using the characteristics aforementioned, we can produce a general model that can be used with any project without prior knowledge about it, eliminating the need to perform the expensive model training step for each project one may want to analyze. This means that the model obtained during the preparation of this paper can be used directly by kiskadee in production to rank warnings in any given project.

To compensate for not using the most relevant characteristics pointed by previous works, we used an ensemble learning method [19] 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 bottom entries were more likely to be false positives.

The results obtained were auspicious: using three static analyzers with a false positive rate of 0.61 when aggregating all three tool warnings in a single report, we achieved an accuracy of 0.8 over the test set. This accuracy was not too distant from the state of the art (0.85 [20]), which depends on characteristics specific to the project being analyzed to train their model. Since our model does not depend on project-specific features, it may receive any project as input without the need to train a different model for each analyzed project.

The following steps describe the techniques and tools used to collect and prepare the data to train our model, as well as the methodology used to train the model and rank the static analysis warnings.

Step 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.

Juliet [6] 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 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 use Juliet version 1.2 to generate our data set.

Before examining Juliet with kiskadee static analysis tools, we pruned the test suite to prevent analysis of test cases that depend on constructs of specific operating systems or external libraries. Table 1 shows the total number of test cases in Juliet before pruning 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.

Step 2: collecting labeled warnings from multiple static analyzers. Based on Juliet 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. Then, we run each static analyzer on the pruned test suite to generate the static analyses reports. Table 2 shows the total number of warnings generated, before discarding warnings whose labeling step cannot be automated based on Juliet documentation. By matching each 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.

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

Table 1: Number of Juliet test cases.

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

Table 2: Warnings generated per tool.

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 3: Labeled warnings per tool.

This set of labeled warnings can finally be examined to extract a training set from it, as discussed next. Table 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 the whole kiskadee report, i.e., the aggregated reports composed of the warnings of all tools.

Step 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 [9, 12, 13, 20]. For instance, Kremenek et al. [13] demonstrate that the tool warning positiveness is highly correlated to code locality.

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.

Step 4: training decision trees with AdaBoost. Given the data collected, we 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 Step 5.

Since we do not post-analyze the source code nor inspect the project history of the analyzed software projects, which are shown to be the best places to look for features to arbitrate on source code static analysis warnings positiveness, 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 with lower error [19].

One widely used ensemble learning method is boosting [19], whose main idea is to run a weak learning algorithm several times in different distributions of the training set to generate and combine various weak classifiers into a stronger one. For this study, we use the AdaBoost algorithm [8], a more general version of the original boosting algorithm [18].

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, 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 [7].

We divide our data set 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. We perform the 10-fold cross-validation technique with different values for **T** (number of weak classifiers trained) in AdaBoost. We then compare the average performance of the classifiers obtained for each distinct 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.

Step 5: ranking static analysis warnings. We use the model trained in Step 4 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 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.

5 Results and Discussion

Although we use a binary classification algorithm to train our model, 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 kiskadee's ranking approach. While comparing our results with other ranking approaches or, at least, with the ranking order of each tool would be ideal, these would not be feasible. In the first case, we would have to replicate other works with

our data set; in the second, it would not make sense to compare the results of a single tool to the aggregate results. Therefore, we chose to compare kiskadee with a random ranking algorithm.

To evaluate our ranking performance over the test set, we refer to the methodology presented by Kremenek et al. [13], which we describe below.

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.

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

It is worth observing that $S\left(R\right) = 0$ for an optimal ranking algorithm and $S\left(R\right) = N_{tp} \times N_{fp}$ 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.

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

$$FP_{avg} = \frac{S\left(R\right)}{N_{tp}} \tag{2}$$

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

$$Performance = \frac{FP_{avg}(random)}{FP_{avg}(\text{AdaBoost ranking})} \tag{3}$$

In a perfect ranking situation, the first false positive occurrence would be positioned after the last true positive occurrence, therefore, $FP_{avg} = 0$. For our test set, in the worst case scenario, one would hit all the false positives before finding the first true positive. Leading to $FP_{avg} = 3942$. When applying a random ranking algorithm, we found $FP_{avg} = 1992$, while, for kiskadee's ranking approach, $FP_{avg} = 380$ over our test set.

The median kiskadee model performance over random, as proposed by Kremenek et al. [13], 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 2 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 for kiskadee.

As Figure 2 shows, kiskadee's 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.

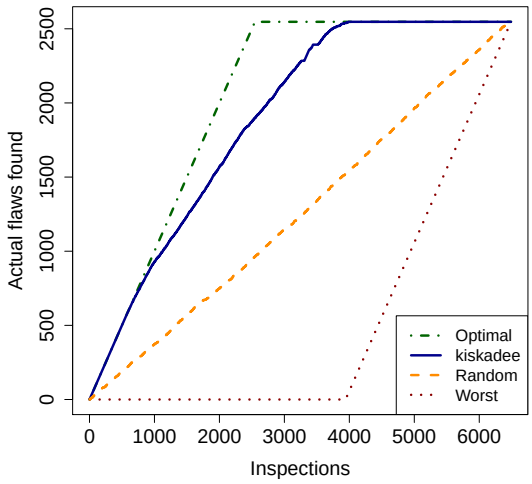

Fig. 2: Number of actual flaws found in top-down inspection of lists ranked with different approaches.

6 Conclusion

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 enable kiskadee to analyze and rank any project given as input, allowing the continuous monitoring and analysis of different software repositories, such as the ones provided by GNU/Linux distributions.

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. Interesting future works include studying confidence levels and the trade-off between loss of information and the cost of inspecting a larger number of false alarms, improving the classification model by collecting users feedback, and investigating other algorithms besides AdaBoost with decision trees as classifiers.

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

References

1. Anitya project website. https://release-monitoring.org, accessed: 2018-01-05
2. Fedora mock-with-analysis project. github.com/fedora-static-analysis/mock-with-analysis, accessed: 2018-01-05
3. Fedora project static analysis special interest group. fedoraproject.org/wiki/StaticAnalysis, accessed: 2018-01-05
4. Firehose mailing list archives. lists.fedoraproject.org/archives/list/firehose-devel@lists.fedoraproject.org, accessed: 2018-01-05
5. Black, P.E.: Static analyzers in software engineering. The Journal of Defense Software Engineering 22(3), 16–17 (2009)
6. Boland, T., Black, P.E.: Juliet 1.1 c/c++ and java test suite. Computer 45(10), 88–90 (Oct 2012)
7. Drucker, H., Cortes, C.: Boosting decision trees. In: Advances in neural information processing systems. pp. 479–485 (1996)
8. Freund, Y., Schapire, R., Abe, N.: A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence 14(771-780), 1612 (1999)
9. Heckman, S., Williams, L.: A model building process for identifying actionable static analysis alerts. In: Software Testing Verification and Validation, 2009. ICST'09. International Conference on. pp. 161–170. IEEE (2009)
10. Heckman, S.S.: Adaptively ranking alerts generated from automated static analysis. Crossroads 14(1), 7 (2007)
11. Hovemeyer, D., Pugh, W.: Finding Bugs is Easy. SIGPLAN Not. 39(12), 92–106 (Dec 2004), http://doi.acm.org/10.1145/1052883.1052895
12. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. Static Analysis pp. 203–217 (2005)
13. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.: Correlation exploitation in error ranking. In: ACM SIGSOFT Software Engineering Notes. vol. 29, pp. 83–93. ACM (2004)
14. Kremenek, T., Engler, D.: Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: International Static Analysis Symposium. pp. 295–315. Springer (2003)
15. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) 1(4), 323–337 (Dec 1992), http://dl.acm.org/citation.cfm?id=161494.161501
16. Muske, T., Serebrenik, A.: Survey of approaches for handling static analysis alarms. In: Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. pp. 157–166. IEEE (2016)
17. Muske, T.B., Baid, A., Sanas, T.: Review efforts reduction by partitioning of static analysis warnings. In: Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on. pp. 106–115. IEEE (2013)
18. Polikar, R.: Ensemble based systems in decision making. IEEE Circuits and systems magazine 6(3), 21–45 (2006)
19. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education, 2 edn. (2003)
20. Ruthruff, J.R., Penix, J., Morgenthaler, J.D., Elbaum, S., Rothermel, G.: Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In: Proceedings of the 30th International Conference on Software Engineering. pp. 341–350. ICSE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1368088.1368135

21. Yoon, J., Jin, M., Jung, Y.: Reducing false alarms from an industrial-strength static analyzer by svm. In: Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. vol. 2, pp. 3–6. IEEE (2014)