# Automated Identification of Actionable Static Code Analysis Alerts on Open Source Systems: Fiware as an Example

**Conference Paper** · November 2018

CITATIONS
0

READS
12

**4 authors**, including:

Ömer Mintemur
Ankara Yildirim Beyazit University
**6** PUBLICATIONS **3** CITATIONS

SEE PROFILE

INTERNATIONAL CONGRESS on
ENGINEERING and ARHITECTURE
14-16 November 2018 Alanya / Turkey

# Automated Identification of Actionable Static Code Analysis Alerts on Open Source Systems: Fiware as an Example

**Ömer MİNTEMUR*[1], Yusuf Şevki GÜNAYDIN[1], Rahime BELEN SAĞLAM[1], Özkan KILIÇ[1]**

[1] Ankara Yıldırım Beyazıt University, Faculty of Natural Sciences and Engineering, Department of Computer Engineering, Ankara/Turkey

*E-mail: omermintemur@gmail.com

## Abstract

With the exponential growth of software systems and the increased dependence on them, it has become a non-trivial and time-consuming task to stabilize the quality of software systems. The research community has started actively developing approaches and tools to detect anti-patterns in order to support developers in planning actions necessary to improve design and source code quality. Static analysis tools examine code for anti-patterns without executing the code, and produce warnings ("alerts") about possible anti-patterns. Despite the common usage of these tools, anti-patterns that they identify are often ignored by developers, since static analysis tools may yield high rates of false positives, which can be very time-consuming to manage. Considering the lost productivity, researchers have been trying to differentiate actionable (must be fixed) from non-actionable (may not be fixed, not a serious enough concern to fix) alerts. We propose an approach to identify actionable alerts by using version history of an open source project with the name FIWARE. Thirteen releases have been covered in the study. We used PMD as a static software analysis tool to analyze the software, and assumed the anti-patterns fixed in the later releases as actionable ones.

**Keywords:** Actionable Alert, Classification, Fiware, Static Code Analysis

## Introduction

Modern people are more dependent on softwares than ever. Since software development is a progressive activity, constantly evolving technology makes software an evolving mechanism. Emergence of software defects is inevitable. Fixing the defects are one of the most important process in software industry. As time passes software sizes have grown substantially which makes though to inspect the software manually. To ease the burden, automatic software analysis (ASA) tools have been widely used in software world to control flow of an software in the means of bugs, and such. Although there are myriad benefits of using ASA programs, they generally produce lots of alerts for a single instance of software. Naturally, as number of alerts increase, challenge for a developer to correct all of these alerts becomes more difficult. It is both time consuming and tiresome activity. For these reasons, researchers have begun to develop techniques to give insights into alerts considering that all the warnings are not equally important. Actionable Alerts require actions (i.e attention) by developers. These kinds of alerts should be interpreted as possible reasons for the software not to function properly in the future. However, non-actionable Alerts, do not require any action by a developer. These kinds of alerts may be interpreted as small, relatively not important warnings. In this study, an open source software called AuthZForce, product of company called FIWARE is analysed using PMD (PMD Source Code Analyzer, 2018) automatic software analysis tool, and actionable alerts is automatically extracted. Results have shown that, this method successfully identified actionable alert and non-actionable alerts. The rest of the papers are organized as follows, in related works, some of the selected works of finding actionable alerts are explained briefly, in material and methods, used software and the method to find actionable alerts are given. Lastly, research findings and discussion is given.

## Related Works

Heckman, S. and Williams, L. used different machine learning algorithms to find Actionable Alerts and also mitigate the false positives which means non-actionable alerts (Heckmans and Williams, 2009). Authors extracted different alert characters from both software and result of static analysis tool used on that particular software. They used alert identifier and alert history, software metrics, source code history, source code churn and aggregate characteristics all of which were extracted from the revision history of a project. After the extraction of alert characteristics (AC), they used different attribute subset evaluation algorithms to select the most important set of attributes using Weka. Three search strategies were employed for selecting AC's: *BestFirst*, *GreedyStepwise*, and *RankSearch*. Then, subset evaluation algorithms were implemented to assess the subsets. Ultimately, two different open source Java projects were utilized to evaluate their model: *jdom*, and *org.eclipse.core.runtime*. The authors evaluated 15 different machine learning algorithms and achieved the best accuracy of %87.8 for *jdom* and accuracy of 96.8% for runtime. They also concluded the following,

- Alert is actionable if the alert was closed in revision history of the software.
- If the alert is closed due to file deletion, the alert is neither actionable or non-actionable and removed from the alert set.
- Other alerts can be classified via inspection or can be marked as non-actionable.

In a different study, three different bug finding tools, namely Findbugs, JLint, and PMD, and software change history were utilized to prioritize warning categories (Kim and Ernst, 2007). The authors basically ranked the warning categories using two software, JEdit and Columba. They also used Kenyon to check out the software transaction in the software change history and marked the time and category of the warning when the warning is appeared in software change history.

Another research proposed a statistical post-analysis based on machine learning methods to classify errors as true or false errors (Yi et. al., 2007). The authors used a program called Airac which is a sound static program analyzer that detects all buffer overrun errors in C programs. The authors constructed their sets by collecting errors from Airac and then they classified those findings true or false manually. After that, they split the data as training and test set and applied 8 machine learning algorithms. According to their results, random forest and boosting generated methods were the most successful classification methods.

Ruthruff et. al. (2008) used statical models which generates binary classifications of warnings. They gathered sample warnings data from Google and proposed a machine learning method which used logistic regression to predict whether FindBugs warning were actionable warning or trivial (non-actionable) warnings. Their method had over 85% accuracy for predicting false positives and over 70% accuracy about finding actionable alerts in a case study performed at Google.

It seems that false positive alerts are an important problem in the field. In another study, Liu K. et. al. (2017) reported that static analysis tools have high false positive alerts rate and defined these false positive alerts as not a serious concern to fix, rarely occur in a runtime environment, or incorrectly defined by a tool. Because of the false alerts raised by some static analysis tools, developers can underestimate these results. Moreover, the authors defined a source code change as a true positive alert. After defining true and false positive alerts, the authors collected 730 open-source Java projects, and they used static analysis tools for each version of projects and gathered fixed and unfixed alerts. To detect if an alert is fixed or unfixed, they emphasized 3 outcomes from previous studies;

- An alert can be dissolved by removing a file or a function enclosing the alert.
- An alert exists at the last revision after tracking means that alert is not fixed yet.
- An alert can be resolved by changing specific lines of code.

According to the previous studies, the first and second outcomes are non-actionable alerts (Hanam et. al. 2014; Heckman and Williams, 2008; Heckman and Williams, 2011) and the third one is accepted as actionable alert (Spacco et. al., 2006; Yi et. al., 2007; Yoon et. al. 2014).

To match potential alerts between revisions they have 3 different approaches;

- location-based matching (looks for a matching alert pair in the same code churn),
- snipped-based matching (match the identical alerts between two revisions),
- hash-based matching (computes the hash value of adjacent tokens of an alert and compares it between two revisions)

In addition to these, Liu K. et. al. tried to define common code patterns for fixed and unfixed alerts with using Convolutional Neural Networks and X-means clustering algorithms. The developers accepted 69 of 116 fixes created from fix pattern of the proposed method and this results showed that this study can help prioritizing violations.

Flynn, L. et al. used different machine learning approaches to prioritize alerts (Flynn et. al., 2018). They collected results from different static analysis tools. Then, authors converted these alerts to CERT Coding Rules by using modified version of tool called SCALe. Then, the authors gave these alert types to auditors for inspection. They employed 4 different machine learning algorithms, namely, Lasso LR, Random Forest, CART, XGBoost on the extracted data used for training. One of the differences of this work from other works is that the authors used experts from the sector which was an important factor for validity. All classifiers achieved over %85 success rate.

**Material and Method**

For the current study, we analyzed AuthZForce, one of the Generic Enablers (GE) of FIWARE. FIWARE is an open source framework which provides API modules, called Generic Enablers, to rapidly develop smart solutions. AuthZForce is used for the authentication of a product. The AuthZForce GE has 13 versions and written in Java programming language. PMD tool which is a source code analyzer was employed in the current study. The versions of AuthZForce were analyzed by PMD. A typical PMD result schema is given in Table 1.

| | |
|---|---|
| Package | Package name which contains the Java file that has errors. |
| File | Java file name which contains errors. |
| Priority | Priority of specific error. Priority levels change in the range of 1 to 5. 1 with the lowest priority, 5 with the highest priority. |
| Line | The erroneous line number. |
| Description | Simple and short explanation of the error. |
| Rule | Rules that fall into specific Rule Set. Each Rule Set has relatively long rule list. |
| Rule Set | Java rules that are predefined by PMD. In this study, all Rule Sets are used. |

**Table 1** - PMD Result Schema

Rule Sets are as follows;

- Best Practices : Rules that are generally accepted as best practice.
- Code Style : Rules that are enforce to use specific code style.
- Design : Rules that help to discover design problems.
- Documentation : Rules that are related to documentation.
- Error Prone : Rules that detect artifact that are prone to errors.
- Multithreading : Rules that deal with multiple threads of execution.
- Performance : Rules that deal with suboptimal code.
- Security : Rules that deal detect security flaws.

After extracting PMD results from the 13 versions' of AuthZForce, we also obtained the function name that each faulty line falls into from the code. Our system works as follows: For two consecutive versions, we compared all the errors in each function. If an error with the same priority and the rule was not detected in the letter version in the same method, it was assumed that the error had been resolved, which indicated that it was actionable. The remaining errors were assumed as non-actionable.

After gathering results, we labeled actionable results as True and non-actionable alerts as False. We selected the features as Line, Function, Priority, and Rule, and used them in K-Nearest Neighbors (KNN) algorithm to classify actionable and non-actionable alerts.
KNN is part of a more general technique known as instance-based learning which uses specific training instances to make predictions for test data by computing the distance between each test example and all the training examples to determine its k nearest neighbors. In the current study, we selected our K value as 5. The classification was done based on the majority class of its nearest neighbors.

By running the algorithm, we successfully classified actionable alert and non-actionable alerts by accuracy of 72% with using relatively small data.

**Research Findings and Discussion**

Research findings showed that 80% of the errors that have been taken action had priority level of 3. Others are 5, 2, and 1 respectively. In all of the errors that were resolved, the most common error was DataflowAnomalyAnalysis which fell into the rule set of Error Prone as seen in Figure 1.

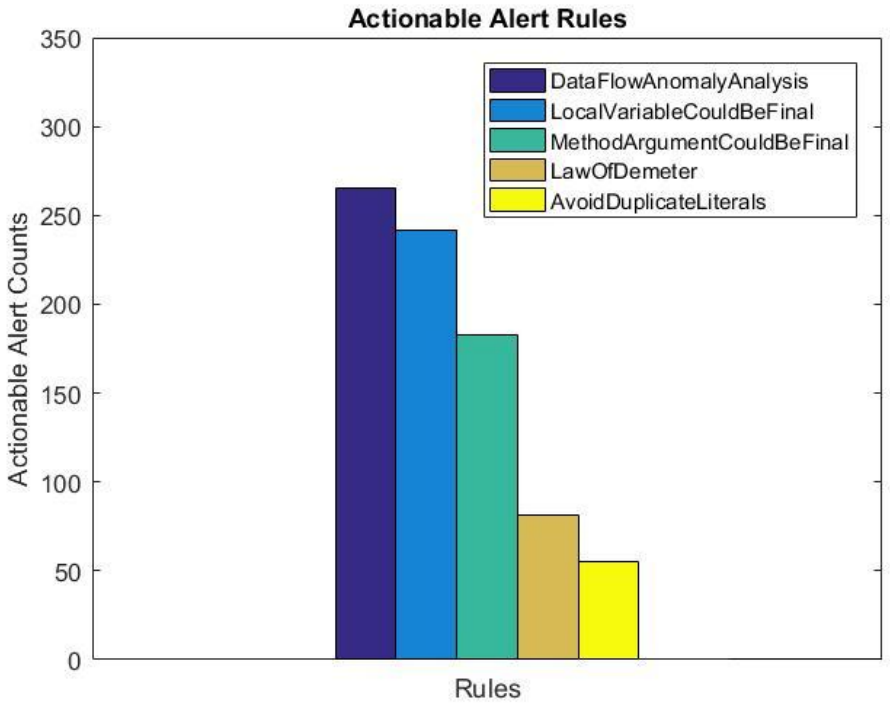


**Figure 1:** Actionable Alert Counts of Rules

The actionable alert numbers can be seen in Figure 2. It has been observed that most of the actionable alerts have been introduced into system in version 8 all of which were resolved in version 9. This could be due to the evolving nature of software. When we analysed the version history of the GE, we noticed that new components were added to the system in version 8 and it yielded an increase in actionable alert count.

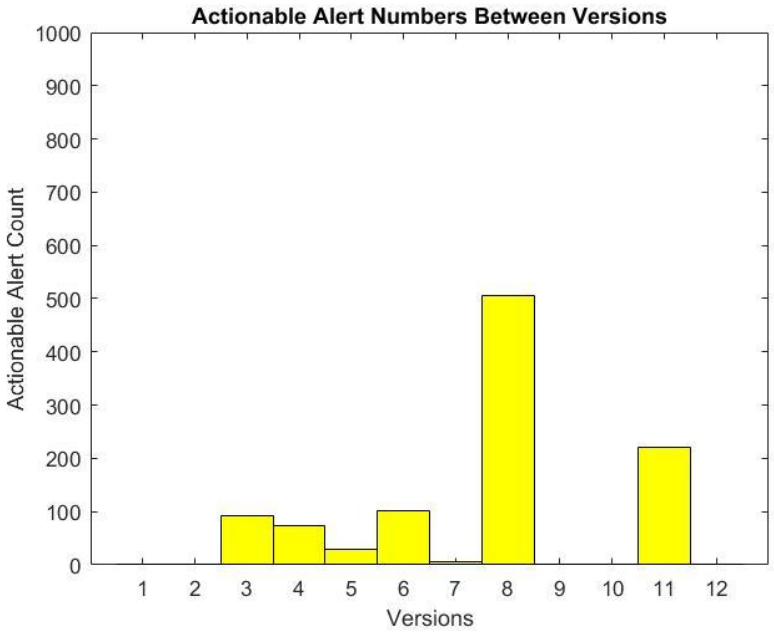


**Figure 2:** Actionable Alert Counts between consecutive versions

Although the results of our work are promising, there are some limitations of the work, which could make generalization of our results harder. The main limitation is labelling the non-actionable alerts. We made the assumption that the errors that had not been resolved within 13 versions were non-actionable. As mentioned before, it is a non-trivial task to define non-actionable alters and there are several approaches in the literature. Even though our assumption has been conducted in several studies in the literature, it might be an oversimplified one for the recent issues for which the developers did not have enough time to take an action. Consequently, recent actionable alerts have the potential to be mislabeled in the training set. In the future, we are planning to assign a waiting time for each type of error before labeling it as non-actionable to relax this oversimplified assumption.

We also would like to analyse the effect of neighborhood among the alerts and to investigate possible tendency to resolve non-actionable alerts that appear in the same file or function with high priority actionable alerts. By observing non-actionable alerts that were resolved by the developers due to its actionable neighbors, it could be possible to work on much trustworthy dataset and to obtain more stable results.

## References

"PMD Source Code Analyzer", https://pmd.github.io/, 2018

S. Heckman and L. Williams, "A Model Building Process for Identifying Actionable Static Analysis Alerts," 2009 International Conference on Software Testing Verification and Validation, Denver, CO, 2009, pp. 161-170.

S. 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). ACM, New York, NY, USA, 2007, 45-54.

Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. 2007. An empirical study on classification methods for alarms from a bug-finding static C analyzer. Inf. Process. Lett. 102, 2-3 (April 2007), 118-123.

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). ACM, New York, NY, USA, 2008, 341-350.

Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., & Traon, Y. L. (2017). Mining Fix Patterns for FindBugs Violations. arXiv preprint arXiv:1712.03201.

S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," in Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. New York, NY, USA: ACM, 2008, pp. 41–50.

Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking," in Proceedings of the 11th Working Conference on Mining Software Repositories. New York, NY, USA: ACM, 2014, pp. 152–161.

S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," Information and Software Technology, vol. 53, no. 4, pp. 363–387, Apr. 2011.

J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking Defect Warnings Across Versions," in Proceedings of the 2006 International Workshop on Mining Software Repositories. New York, NY, USA: ACM, 2006, pp. 133–136.

J. Yoon, M. Jin, and Y. Jung, "Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM," in 2014 21st AsiaPacific Software Engineering Conference, vol. 2, Dec. 2014, pp. 3–6.

K. Yi, H. Choi, J. Kim, and Y. Kim, "An empirical study on classification methods for alarms from a bug-finding static C analyzer," Information Processing Letters, vol. 102, no. 2–3, pp. 118–123, Apr. 2007.

Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. 2018. Prioritizing alerts from multiple static analysis tools, using classification models. In Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies (SQUADE '18). ACM, New York, NY, USA, 13-20.