2013 IEEE International Conference on Software Maintenance

Automated Classification of Static Code Analysis Alerts: A Case Study

Ulaş Yüksel*†, Hasan Sözer†
*Vestel Electronics, Manisa, Turkey
†Özyeğin University, İstanbul, Turkey
{ulas.yuksel, hasan.sozer}@ozu.edu.tr

***Abstract*—Static code analysis tools automatically generate alerts for potential software faults that can lead to failures. However, developers are usually exposed to a large number of alerts. Moreover, some of these alerts are subject to false positives and there is a lack of resources to inspect all the alerts manually. To address this problem, numerous approaches have been proposed for automatically ranking or classifying the alerts based on their likelihood of reporting a critical fault. One of the promising approaches is the application of machine learning techniques to classify alerts based on a set of artifact characteristics. In this work, we evaluate this approach in the context of an industrial case study to classify the alerts generated for a digital TV software. First, we created a benchmark based on this code base by manually analyzing thousands of alerts. Then, we evaluated 34 machine learning algorithms using 10 different artifact characteristics and identified characteristics that have a significant impact. We obtained promising results with respect to the precision of classification.**

***Keywords*—alert classification, industrial case study, static code analysis**

I. Introduction

Static code analysis tools (SCATs) inspect the source code of programs to automatically pinpoint (potential) faults without actually executing these programs [1]. They complement manual software reviews and testing activities to assist in the development of reliable software systems. The main disadvantage of SCATs is the large amount of alerts (i.e., warnings, issues) being exposed to developers. The density of alerts can be as much as 40 alerts per thousand lines of code (KLOC) [2]. Although 2 alerts per KLOC on average is typical in our experience, even this alert density leads to an information overload. Around 3000 alerts are generated for an industrial scale software of size 1500 KLOC. Moreover, these alerts are subject to false positives. Empirical results have shown that effective SCATs have false positive rates ranging between 30% and 100% [3]. Hence, every alert should be (manually) inspected by developers to identify which alerts should be considered (i.e., true positives, alerts that are actionable [2]) and which should not. This process requires considerable time and effort such that the cost of using SCATs overcomes its benefits. If we assume the inspection time to be 5 min. per alert [2], [4], a developer might need up to 250 hours of time to inspect 3000 alerts.

We have observed that the inspection of SCAT alerts constitute a significant amount of time and effort during the software development lifecycle. The problem grows as we recognize a saturation effect of high number of alerts when the software evolves. High ratio of false positives also makes the use of SCATs discouraging. SCATs often provide a prioritization (i.e., severity measure) for the reported alerts. Sometimes this information is consulted to focus on the highly prioritized alerts only. The majority of the alerts might be ignored this way. Although this approach can save some time and effort, one can miss the opportunity to fix some important defects in early development stages.

To address this problem, the output of SCATs can be (automatically) post-processed to classify alerts and eliminate false-positives. Numerous approaches have been proposed for automatically classifying/ranking the alerts of SCATs based on their likelihood of reporting a critical fault [5]–[10]. One of the promising approaches is the application of machine learning techniques to classify alerts based on a set of artifact characteristics [6]. In this work, we evaluate this approach in the context of an industrial case study to classify the alerts generated for a digital TV software. The system comprises millions of lines of code that has evolved for more than a decade. We created a benchmark based on this code base by manually analyzing thousands of alerts. The benchmark is used for 3 different studies. First, 10 different artifact characteristics are evaluated for identifying the most relevant attributes for classification. Second, 34 different machine learning algorithms are evaluated with respect to the accuracy, precision and recall measures. Third, several different test sets are used for evaluation, each of which reflects a different point of time (i.e., snapshot) during the software development life cycle. We obtained promising results with respect to the precision of classification, which can range from 80% up to 90%.

The remainder of this paper is organized as follows. The following section presents the industrial case. In Section III, we describe the data set and characteristics we analyzed for alert classification. Section IV discusses the case studies and results. Related studies are summarized in Section V. Finally, conclusions are provided in Section VI.

II. Industrial Case: Digital TV

In this section, we introduce an industrial case for classifying alerts generated for a Digital TV software system. This is an embedded soft real-time system and it has been maintained by Vestel Electronics[1], which is one of the largest TV manufacturers in Europe. The software was first developed for digital video broadcasting (DVB) set-top boxes by Cabot Communications[2] founded in 1993. In 2001, the company

[1]http://www.vestel.com.tr.
[2]http://www.cabot.co.uk.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.89

was acquired by Vestel Electronics and the software has been extended to support both digital and analog TV applications involving many different features such as video recording, media browsing and web browsing. During this time frame, hundreds of software developers have performed tens of thousands of changes resulting in a software system with over 1500 KLOC distributed in over 30000 classes/types and over 8000 files.

The software is composed of three layers: *i) application*, *ii) middleware* and *iii) driver*. The *application layer* controls the behaviour of the middleware layer. It implements an on-screen menu system and manages user interaction through remote control, front panel keys and other control interfaces.

The *middleware layer* implements the main features including service installation, date/time handling, audio and video playing, teletext/subtitle display and software updates. Both the middleware and the application layers have an object-oriented design, and they are mainly implemented in the C++ language.

Middleware is ported to different hardware platforms using a uniform interface provided by the *driver layer*. This is an interface that is implemented in the C language and it provides access to operating system primitives as well as hardware device drivers. The driver layer comprises external libraries that are developed by hardware suppliers. Hence, in our case study, we focused on the application and middleware layers only. We have the complete source code and version history for these layers. In the following section, we summarize the artifact characteristics and the data set utilized for analyzing this code base.

III. Analyzed Artifact Characteristics and the Data Set

In Vestel, a dedicated bot server regularly checks out the project source code from an Subversion (SVN) server and runs a commercial SCAT[3]. For this study, we took the first 19 weekly run of SCAT over the code base in SVN. SCAT generates a list of alerts with a unique *ID* and a set of characteristics such as *severity*, *alert code*, *alert state* (*open* or *fixed*), *file name*, *method name*, and *line number*.

Software developers can inspect the generated alerts and provide feedback about them. We call this feedback the *developer idea* and we consider it as a characteristic for classifying SCAT alerts. We have selected in total 10 different characteristics to classify alerts. These characteristics are mainly pointed out in previous similar studies [2], [7], [9] although the *developer idea* has not taken much attention in the literature except only a few studies [5]. The analyzed characteristics are listed in the following.

Severity: An ordinal number between 1 and 4, which is assigned by SCAT for indicating the importance of the problem (1 is the highest, 4 is the lowest).

Alert code: An abbreviation assigned by SCAT, which indicates the type of alert. e.g., *NPD* stands for *Null Pointer Dereferencing*.

Lifetime: Number of periodic SCAT runs between the first discovery and the closure of an alert, if the alert is closed; number of SCAT runs between the first discovery and the current time, if the alert is still open.

Developer idea: One of the four different values: *i) fix*, *ii) not a problem*, *iii) ignore*, and *iv) analyse*. The default value is *analyse*. It becomes *fix* if the developer conforms and fixes the issue reported by the alert. At the next run of SCAT, such alerts are closed automatically if the issue is really resolved. If the developer thinks that the alert is a false positive, the *developer idea* is set as *not a problem*. If the alert is not a false positive, yet the reported issue is not considered important/relevant, the *developer idea* is set as *ignore*. An alert can also be fixed and closed implicitly during the development or a bug fix process without any alert inspection. In that case, the *developer idea* keeps the value *analyse* but the *alert state* changes to be *fix* silently at the next run of SCAT.

File name: Source file name.

Module name: Root folder name.

Open alerts: Number of open alerts, if any.

Total alerts: Number of alerts, if any.

Total alerts in module: Number of alerts in a particular module, if any.

Total alerts in file: Number of alerts in a particular file, if any.

After collecting data regarding the characteristics listed above, we have manually inspected each alert associated with the source code and determined if the alerts generated by SCAT is actionable or unactionable (i.e., false positive). As such, we created a full oracle that we use in our analysis with 1147 total alerts (498 actionable and 649 unactionable). Our inspection results are listed in Table I.

TABLE I. Mapping of alert inspection and resolution.

		Manual Inspection Result		
		True Positive	False Positive	Total
Resolution	Resolved	203	62	265
	Not resolved	295	587	882
	Total	498	649	1147

During the 19 weekly period of SCAT runs, some alerts were evaluated and resolved by developers independent from our study. 265 alerts are resolved in which 62 of them are false positives according to our manual inception. We observed that developers mainly tend to consider the *severity* measure provided by SCAT during the selection of alerts to be resolved. Almost 80% of the resolved alerts have the highest *severity* value. On the other hand, 882 alerts have remained unresolved, where 295 of them are true positives according to our inspection.

IV. The Case Study and Results

In this section, we first summarize the 3 different studies we performed. Then we present our analysis in detail and discuss the results.

In the first study, we evaluated the relevance of the 10 selected characteristics as attributes for alert classification. For

[3]We keep the name of the tool confidential.

this purpose, we used 10 different attribute evaluator tools from the Weka (Waikato Environment for Knowledge Analysis) toolset [11]. Weka is an open source toolset, which supports common data mining features such as classification, clustering, regression, association, and attribute selection based on various machine learning algorithm implementations. In the second study, we used the full data set (i.e., alerts generated throughout the 19 runs of SCAT) and 10-folds cross-validation [11] to evaluate the accuracy of different machine learning algorithms. We tested in total 34 different machine learning algorithms implemented in Weka. In the third study, we used a training set based on the alerts generated until the end of the 5^{th} run of SCAT only. Then, we classified alerts generated in later phases of the project. In the following, we explain our studies and the corresponding analysis in detail, together with our results.

a) Attribute Evaluation: In this study, we aimed to see which of the 10 selected characteristics are evaluated to be relevant for alert classification. We collected all the alerts generated for our case. We used 10 different attribute selection evaluator tools of Weka. 7 of these tools provide a merit and ranking for each of the evaluated attributes. The other 3 just perform a binary evaluation and select a sub-set of the attributes. For these 3 tools, we assigned a merit to each attribute based on the cardinality of the sub-sets. (e.g., if only 4 characteristics are selected out of 10, we assigned value 2.5 to the selected and 7.5 to the unselected ones.) Then, we calculated the mean value of merits obtained/derived from the 10 different tools for each characteristic. Figure 1 presents the results both in terms of the mean ranking value and the number of times a characteristic appears in top four ranks. Hereby, *file name, lifetime, alert code, developer idea* and *severity* appear to be the most relevant characteristics for classification.

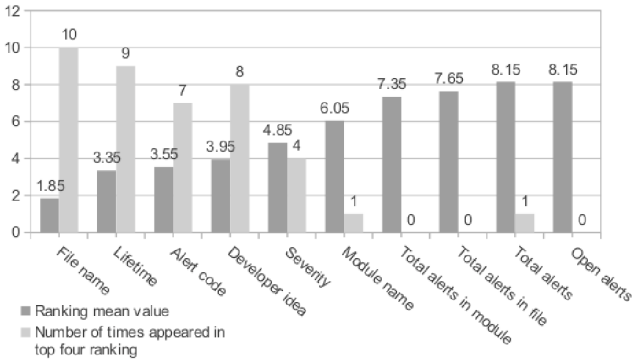

Fig. 1. Artifact characteristics ranking results (ordered).

b) Classification with 10-folds cross validation: In the second study, we evaluated the effectiveness of classification by applying 10-folds cross validation [6], [12] over the whole set of the generated alerts. We tested 34 different machine learning algorithms - with default settings - implemented in Weka. We have observed that the accuracy ranges between 45% and 86%. Table II lists the classifiers having the best accuracy values based on the initially selected 10 characteristics.

c) Classification during the development life-cycle: 10-folds cross-validation uses the full data set for both the training set and the test set. As a result, the test set includes exact values for the *lifetime* and the *developer idea* characteristics.

TABLE II. TOP CLASSIFIERS WITH RESPECT TO ACCURACY.

Classifier	Accuracy	Precision[a]	Recall[a]
Random forest [13]	86.1%	86.1% [0.84,0.88]	86.1% [0.84,0.88]
Random committee	86.4%	86.4% [0.85,0.88]	86.4% [0.84,0.88]
DTNB [14]	83.6%	83.8% [0.80,0.87]	83.6% [0.84,0.84]

[a] Values in brackets are particular calculations for true positive and false positive classifications respectively.

In fact, these values would not be available for alerts that are just generated by the latest SCAT run. Hence, in our third study, we took *snapshots* at certain SCAT runs. We prepared incremental test sets at these snapshots to mimic a real case during the development life cycle. We took the alerts that are generated during the first 5 SCAT runs. The training set that is prepared based on these alerts represents 92.94% of the full data set. This is due to fact that a legacy software has been used, for which the first run of SCAT already generated a high number of alerts. If an alert in the training set was resolved after the 5^{th} run of SCAT, then we recalculated the *lifetime* value instead of using the recorded *lifetime*. In addition, we reset the *developer idea* as *analyze*, because at that point in time, this alert had not been investigated and resolved yet. Then, we created 3 test sets from the remaining alerts. These sets can be seen in Table III, where the alerts are grouped according to *SCAT run number* for the SCAT run they first appear (e.g., SCAT run # [6,10]) We reset all the *developer idea* and *lifetime* characteristics in these test sets too, as if they were new and had not been considered yet.

TABLE III. TEST SETS REFLECTING DIFFERENT POINTS OF TIME DURING THE DEVELOPMENT LIFE CYCLE.

Group	SCAT run #	True Positive	False Positive	Total
1^{st}	[6,10]	2	13	15
2^{nd}	[11,16]	9	11	20
3^{rd}	[17,19]	34	12	46

We trained the 34 machine learning algorithms with our training set (based on SCAT run # [1,5]) and then evaluate them with respect to the 3 test sets (Table III). Table IV presents the results for one of the classifiers (DTNB), which was one of the most accurate classifiers according our second study (Table II). We can observe a significant improvement from group 1 to group 2, and from group 2 to group 3, as well. This might be explained by the increasing number of true positives involved in the test sets (See Table III), while the number of false positives are almost equal. In general, the results turned out to be promising; the average accuracy, precision and recall is around 90% for the third group.

TABLE IV. DTNB CLASSIFICATION RESULTS FOR DIFFERENT POINTS OF TIME DURING THE DEVELOPMENT LIFE CYCLE.

Group	Accuracy	Precision[a]	Recall[a]
1^{st}	66.7%	80.7% [0.20,0.90]	66.7% [0.5,0.69]
2^{nd}	80.0%	86.2% [0.69,1.00]	80.0% [1.00,0.64]
3^{rd}	89.1%	90.7% [0.97,0.73]	89.1% [0.88,0.92]

[a] Values in brackets are particular calculations for true positive and false positive classifications respectively.

V. Related Work

There exist several case studies performed to reveal the capabilities and limitations of SCATs. Zheng et al. collect experiences from the development of a set of industrial software products. In their study, SCAT alerts are compared with respect to manual inspection results and faults found during tests. They conclude that SCATs constitute an economical value, being complementary to other verification and validation techniques [15]. Boogerd et al. [16] present an industrial case study for identifying correlations between faults and violations detected regarding the coding standard rules. They conclude that true positive rates may highly differ from project to project.

Some other studies focus on improving the usefulness of SCATs by ranking or classifying alerts [2]. An adaptive ranking model [5] is proposed for adjusting the ranking factor of each alert type based on its statistical contribution to the previous rankings and the feedback of developers for the previously ranked alerts. A benchmark [17] is introduced based on a set of Java programs for comparing classifiers with respect to their effectiveness. In particular, 3 Java programs are used for testing different machine learning algorithms with a wide range of artifact characteristics [6] to identify the most effective subset(s) of characteristics for alert classification. Kim and Ernst propose an approach for exploiting "historical information" to prioritize alerts [7]. Hereby, they mine the software change history for alerts that are removed during bug fixes. Alert categories are prioritized based on fixes applied on the corresponding alerts. Liang et al. also employ bug fix history for alert classification [9] and differentiate bug fixes as either "project-specific" or "generic". They conclude that one may improve precision by using history regarding the "generic" bug fixes only.

Previous studies make use of a single set of alerts over the whole project life time. In this work, we took several *snapshots* from the alert history and created different training/test sets, in which artifact characteristics (e.g., *lifetime*) reflect the corresponding point of time during the software development life cycle. As another difference, previous related studies mostly focus on Java programs, whereas we work on embedded software and a C/C++ code base. To the best of our knowledge, the use of machine learning techniques for classifying SCAT alerts have not been evaluated in the context of industrial case studies for this application domain before.

VI. Conclusion and Future Work

In this work, we evaluated the application of machine learning techniques to automatically classify SCAT alerts. We created a data set by manually inspecting thousands of alerts generated for a digital TV software. Then, we performed 3 different studies for *i)* ranking the relevance of 10 different characteristics by 10 different attribute evaluators, *ii)* evaluating 34 different classifiers with 10-folds cross validation, and *iii)* evaluating the effectiveness of classification at different points of time during the software development life-cycle. The characteristics *lifetime*, *developer idea*, *file name*, *alert code* and *severity* were identified as the most relevant attributes. We obtained promising results with respect to the precision of classification, which can go up to 90%. Hence, we conclude that the use of machine learning techniques can be a viable approach for automated classification of SCAT alerts.

In this study, we created a benchmark by using the output of the first 19 weekly runs of the SCAT. In our future work, we plan to extend our data set with the consequent SCAT runs of the same software project. We also plan to incorporate additional artifact characteristics based on our observations. For instance, the *developer idea* is a subjective attribute that can take very different values based on the insights, experience, sensitivities and ambitions of different developers. Therefore, the *developer idea* can be combined with the *developer name* to increase the precision of classification.

References

[1] R. Tischer, R. Schaufler, and C. Payne, "Static analysis of programs as an aid for debugging," *SIGPLAN Notices*, vol. 18, no. 8, pp. 155–158, 1983.

[2] 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, 2011.

[3] T. Kremenek and D. Engler, "Z-ranking: using statistical analysis to counter the impact of static analysis approximations," in *Proceedings of the 10th international conference on Static analysis*, 2003, pp. 295–315.

[4] "Effective management of static analysis vulnerabilities and defects," White Paper, Coverity Inc., 2009.

[5] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *Crossroads*, vol. 14, no. 1, p. 7:17:11, Dec. 2007.

[6] ——, "A systematic model building process for predicting actionable static analysis alerts," Ph.D., North Carolina State University, United States – North Carolina, 2009.

[7] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, p. 27.

[8] ——, "Which warnings should I fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, p. 4554.

[9] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, p. 93102.

[10] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, "A framework to compare alert ranking algorithms," in *2012 19th Working Conference on Reverse Engineering (WCRE)*, Oct. 2012, pp. 277–285.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[12] 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. 23, pp. 118–123, 2007.

[13] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, 3rd ed. Morgan Kaufmann, Jan. 2011.

[14] M. A. Hall and E. Frank, "Combining naive bayes and decision tables," Florida,USA, 2008, pp. 318–319.

[15] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, Apr. 2006.

[16] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, Oct. 2008, pp. 277 –286.

[17] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, p. 4150.