A Measurement Framework of Alert Characteristics for False Positive Mitigation Models

Sarah Heckman and Laurie Williams
North Carolina State University
sarah_heckman@ncsu.edu and *williams@csc.ncsu.edu*

Abstract
Automated static analysis tools can be used to identify potential source code anomalies early in the software process that could lead to field failures. However, only a small portion of static analysis alerts may be important to the developer (actionable). The remainder are false positives (unactionable). Static analysis tools may generate an overwhelming number of alerts, the majority of which are likely to be unactionable. False positive mitigation techniques utilize information about static analysis alerts, called alert characteristics, to predict actionable and unactionable alerts. This paper presents a measurement framework for generating static analysis alert characteristics for false positive mitigation models.

1. Introduction
Automated static analysis tools can be used to identify potential source code anomalies early in the software process that could lead to field failures. Inspection of each alert by a developer is required to determine if the alert is an indication of an important anomaly that the developer wants to fix, which we call a true positive (TP) or an actionable alert [1, 15]. When an alert is not an indication of an anomaly or is deemed unimportant to the developer (e.g. the alert indicates a programmer mistake inconsequential to program functionality), we call the alert a *false positive* (FP) or an *unactionable alert* [1, 15].

Static analysis tools may generate an overwhelming number of alerts [9], the majority of which are likely to be unactionable [6]. FP mitigation techniques utilize information about the alerts, called alert characteristics (AC), to prioritize alerts by the likelihood of being actionable or to classify alerts into actionable and unactionable groups [3, 8-10, 15, 16]. For any FP mitigation model, we want to extract two types of useful information from static analysis alert data: 1) sets of ACs that are predictive of actionable alerts; and 2) which models (using the predictive ACs) are best at classifying alerts as actionable or unactionable. Prior research [3, 8-10, 15, 16] has shown that several ACs are predictive of actionable or unactionable alerts. We build on prior research by creating a measurement framework for 51 candidate alert characteristics that may be used to build FP mitigation models.

The rest of this paper is organized as follows: Section 2 provides related work on ACs used in other FP mitigation techniques, Section 3 describes the sources for ACs; Sections 4-8 discuss the five categories of ACs and how each AC is generated in detail. Section 9 concludes with how the AC measurement framework can be used.

2. Related Work
This section describes the ACs used by other FP mitigation techniques. We include most of these ACs in this work in addition to a few others.

Our prior research [3, 4] has proposed a project-specific, in-process, FP mitigation prioritization technique that utilizes the *alert's type* and *location at the source folder, class, and method levels*. The model, AWARE-APM [3, 4], also uses developer feedback in the form of *alert suppression* and *alert closures*. Suppressing an alert is an explicit developer action to indicate the alert is unactionable. Closure is determined by comparing subsequent static analysis runs. If the alert is not in the later run, the alert is closed. After a developer inspects the alert and takes an action on that alert, the prioritization of the remaining alerts is adjusted from the feedback. We evaluated three versions of AWARE-APM model on the FAULTBENCH benchmark subject programs and found an average accuracy of 67-76% [3]. The precision and recall were in the 16-19% and 25-42% range, respectfully, for the benchmark programs. The low accuracy suggests that while the models may work well for some programs, the models do not work well for others. Additionally, the alert type and alert location together and in isolation may not be the best predictors of actionable alerts.

Ruthruff et al. [15] screened 33 ACs from 1,652 alerts sampled from Google's code base to develop logistic regression models for predicting actionable and unactionable alerts. Ruthruff et al. describe a screening process whereby ACs were selected for the model. The generated models contained 9-15 ACs and had an accuracy ranging from 71-87%. Ruthruff et al. [15] compared their generated models to a linear regression model containing all ACs and models developed by Bell et al. [2, 13] for predicting the number of faults. Overall, the models generated by Ruthruff et al. generally had a higher accuracy than the other models. Additionally, the time to gather the data to build the generated model was substantially shorter than the time to build the model with all ACs. Many of the ACs suggested by Ruthruff et al. are used in our research in addition to other project specific metrics. We also consider additional machine learners.

Kim and Ernst [8, 9] describe two static analysis alert prioritization techniques that utilize data mined from source code repositories. The first prioritization technique uses the average lifetime of alerts sharing the same type to prioritize the alert types [8]. The lifetime of an alert is the time (in days) between alert creation and alert closure. Kim and Ernst assumed that alert types with shorter lifetimes have a higher ranking (e.g. alerts fixed quickly are likely important).

The second technique is a history-based alert prioritization that weights alert types by the number of alerts closed by fault- and non-fault-fixes. A fault-fix is a source code change where the developer fixes a fault or problem and a non-fault-fix is a change where a fault is not fixed, like a feature addition [9]. Alerts may be closed during any code modification, and are therefore considered actionable, but Kim and Ernst expect that those alerts closed during fault-fixes are more important when predicting actionable alerts.

The history-based alert prioritization presented by Kim and Ernst [9] improves the alert precision by over 100% when compared to the alert precision of alerts prioritized by tool severity. However, the precision ranged from 17-67%, which might be due to alert closures not having causal relationships with the root cause of an anomaly-fix. We include the alert lifetime, measured in revisions instead of days, as a candidate AC. We also utilize source code repository mining for other ACs. Unlike Kim and Ernst, we are interested in prioritizing or classifying individual alerts rather than the alert type.

Williams and Hollingsworth [16] created a static analysis tool which evaluates how often the return values of method calls are checked in source code. A method is flagged with an alert when the return value for the method is inconsistently checked in calling methods. Williams and Hollingsworth use the HISTORYAWARE prioritization technique to prioritize methods by the percentage of time the return value for the methods are checked in the software repository and the current version of the code. The results show a FP rate of 70% and 76% when using the HISTORYAWARE prioritization technique on two case studies involving httpd[1] and Wine[2] applications, respectively. The HISTORYAWARE technique mines data from the source code repository, which we also do, but for different ACs. Instead of using alert type specific information to identify actionable alerts, we use ACs that can prioritize or classify many alert types.

Kremenek et al. [10] show that static analysis alerts in similar locations tend to be homogeneous. On average, 88% of methods, 52% of files, and 13% of directories with two or more alerts contained homogeneous alerts. Kremenek et al. created a FEEDBACK-RANK algorithm whereby the developer's feedback is used to prioritize the remaining alerts. The static analysis tools used by Kremenek et al. take advantage of understanding where a static analysis tool checked for an alert, but did not find a potential anomaly [11]. Kremenek et al. [10] prioritize the alerts via a Bayesian Network [17].

3. AC Sources

There are three potential sources for static analysis ACs: static analysis tools, source code metrics tools, and source code repositories. Most static analysis tools, like FindBugs [6], provide identifying information about an alert like the location, type, priority, etc. A static analysis tool may only provide a listing of high priority alerts or of specific alert types (e.g. null pointer alerts rather than style alerts).

Nagappan et al. [12] show that code complexity metrics correlate with failure-prone modules. Additionally, Bell et al. [2, 13] have utilized code size metrics to predict fault counts. Actionable alerts could be considered faults; therefore, software metrics could be predictive of actionable alerts. There are many tools that generate metrics at the file, package, and project levels like JavaNCSS[3] or Metrics v1.3.6[4]. These tools provide information about the size of source code by lines and the complexity [5] of the programs (e.g. cyclomatic complexity, depth of inheritance, etc.).

[1] http://httpd.apache.org/
[2] http://www.winehq.org/
[3] http://www.kclee.de/clemens/java/javancss/
[4] http://metrics.sourceforge.net/

The models by Williams and Hollingsworth [16], Kim and Ernst [8, 9], and Ruthruff et al. [15] use ACs obtained from a project's source code repository to predict actionable alerts. Using the source code repository allows us to determine how the set of alerts generated by static analysis and the code base has changed over time: using the past to predict the future.

We categorize the ACs into five groups, which are discussed in Sections 4-8: alert identifiers and alert history, software metrics, source code history, source code churn, and aggregate characteristics. In some cases, there are ACs that are collected that are not used in prediction models but that are instead used as part to generate other metrics. The ACs not used in prediction models will be signified with an asterisk (*). Additionally, references will be provided where these ACs have been used in related work.

4. Alert Identifiers and Alert History

This section discusses the alert identifiers generated by static analysis tools and the alert history determined by analyzing alerts generated across the history of the project.

4.1. Alert Identifiers

A static analysis tool generates *alert identifiers* at alert creation. Alert identifiers are typically generic across static analysis tools and provide information about the type of alert and where the alert is located. Specifically, the alert identifiers generated by FindBugs [6] are presented below.

- **Project name:** the name of the project under static analysis. For our research a project can be loosely defined as a logical grouping software, which is "computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system" [7].
- **Package name:** In Java, the name of the package containing the file which contains an alert. The package name could also be generalized to the path starting at the project to the file containing an alert [3, 4, 10].
- **File name:** The name of the source file containing an alert [3, 4, 10].
- **Method signature:** The name and parameter types of the method or function containing the alert (e.g. methodName(String, Object)) [3, 4, 10]. An alert may not have an enclosing method (e.g. the alert is on an instance or global variable).
- **Alert type:** The type of potential anomaly (e.g. null pointer, etc.) as defined by the static analysis tool [3, 4, 8, 9, 15, 16].
- **Alert category:** A high level categorization of alert types (e.g. security, correctness) as defined by the static analysis tool [15].
- **Priority:** The priority of the alert defined by a static analysis tool [8, 15].
- **File extension:** The extension of the file containing the static analysis alert [15]. Ruthruff et al. [15] were able to use the file extension to differentiate between files generated during project builds.
- **Description*:** Provides a description of the alert for the developer to read to help determine if the alert is actionable. The description is tool specific.
- **Identifier*:** An identifier for the alert generated by the tool. FindBugs [6] provides an instance hash for each alert.
- **Line Number*:** The line containing the alert. If the alert spans more than one line, the line number is typically the first line.

4.2. Alert History

The alert history is generated by iteratively going through source code revisions, starting with the earliest revision, to determine alert creation and closure. A *revision* is a set of changes committed to the source code repository together. The AC measured is the number of times that an alert changes over the history of the alert. We consider an alert modification to be when an alert's line number, identifier, or tool generated priority are changed over time. An alert is considered the same alert if the project name, package name, file name, method signature, alert type, and either of the identifier or line number is the same.

5. Software Metrics

Software size and complexity metrics have been used to predict fault- and failure-prone software [2, 12, 13], and could be useful for predicting actionable and unactionable alerts. The metrics outlined below come from the JavaNCSS metrics tool. Other metrics tools or a combination of metrics tools could provide additional metrics at the expense of increased runtime.

- **Method Size:** The number of non-comment source statements (NCSS) within the method containing the alert. If the alert is not within a method, then the method size is set to -1.
- **File Size:** The number of non-comment source statements (NCSS) within the file [15] containing the alert. If the alert is not within a file, then the file size is set to -1.
- **Package Size:** The number of non-comment source statements (NCSS) within the package containing the alert. If the alert is not within a package, then the package size is set to -1.

- **Number of Methods in File:** The number of method declarations within the file containing an alert.
- **Number of Classes in File:** The number of class declarations within the file containing an alert. There could be more than one class in a Java file if the file contains inner classes.
- **Number of Methods in Package:** The number of method declarations within the package containing an alert.
- **Number of Classes in Package:** The number of class declarations within the package containing an alert.
- **Cyclomatic Complexity:** Measures the number of paths through a method [14] containing an alert. Ruthruff et al. [15] use indentation as a measure of complexity.

6. Source Code History

The source code history provides a record of how a project has evolved over time. We can use the source code history to find important events in an alert's lifetime. An alert is *created* or *opened* if the alert is not in any of the prior revisions [3, 4]. An alert closure occurs when the alert was in a prior revision, but is not reported in a later revision [3, 4]. An alert is reopened if the alert was closed in a prior revision and available in a later revision. Additionally, the source code history can reveal other important events that may influence if an alert is actionable or unactionable.

- **Alert Open Revision:** The revision an alert is first opened [8].
- **Alert Close Revsion*:** The latest revision an alert is closed if the alert is closed.
- **Developers:** The set of developers who made changes to the file containing an alert between prior revision analyzed and the alert's open revision [8].
- **File Creation Revision:** The revision a file is first created [15].
- **File Deletion Revision**: The latest revision the file no longer exists in. If the file is re-created at a later revision, the file deletion revision is set to -1. Alerts closed due to a file deletion are not considered actionable [8, 9, 15]. These alerts are removed if the file deletion revision is less than or equal to the closure revision. We can obtain the file deletion revision at the exact revision of deletion, but if a subset of revisions is analyzed for static analysis alerts, the alerts closure revision may not be the same as the file deletion revision.
- **Latest File Modification Revision**: The last modification to a file containing an alert before the last analyzed revision [15]. This AC provides information about the latest changes that may determine if an uninspected alert is actionable or unactionable.
- **Latest Package Modification Revision**: The last modification to a package containing an alert before the last analyzed revision [15]. This AC provides information about the latest changes that may determine if an uninspected alert is actionable of unactionable.
- **Latest Project Modification Revision**: The last modification to a project containing an alert before the last analyzed revision [15]. This AC provides information about the latest changes that may determine if an uninspected alert is actionable of unactionable.

7. Source Code Churn

Source code churn measures the amount of change made to a file, package, or project over time [15]. We are specifically interested in the changes that occurred at the file, package, and project level that may have caused an alert to be created. Therefore, we measure the source code churn that occurred on and before the open revision for an alert. Specifically, we are interested only in the changes that occurred between the last analyzed revision of software and the alert's open revision. Source code repositories like CVS[5] or SVN[6] record churn metrics at each commit. If there is no repository for a project, then a diff utility[7] may be used.

- **File Added lines:** The number of lines added to a file that were not there before [15]. These lines can include comments and white-space.
- **File Deleted lines:** The number of lines deleted from a file that were there before [15]. These lines can include comments and white-space.
- **File Growth:** The difference between added and deleted lines for a file [15].
- **File Total modified lines:** The sum of added and deleted lines for a file [15].
- **File Percent modified lines:** Percent of file total modified lines out of all churned lines for the project [15].
- **Package Added lines:** The summation of all file added lines for files in the same package [15]. These lines can include comments and white-space.
- **Package Deleted lines:** The summation of all file deleted lines for files in the same package [15]. These lines can include comments and white-space.
- **Package Growth:** The difference between added and deleted lines for a package [15].

[5] http://ximbiot.com/cvs/wiki/
[6] http://subversion.tigris.org/
[7] http://www.gnu.org/software/diffutils/diffutils.html

- **Package Total modified lines:** The sum of added and deleted lines for a package [15].
- **Package Percent modified lines:** Percent of package total modified lines out of all churned lines for the project [15].
- **Project Added lines:** The summation of all package added lines for packages in the same project [15]. These lines can include comments and white-space.
- **Project Deleted lines:** The summation of all package added lines for packages in the same project [15]. These lines can include comments and white-space.
- **Project Growth:** The difference between added and deleted lines for a project [15].
- **Project Total modified lines:** The sum of added and deleted lines for a project [15].
- **Project Percent modified lines:** Percent of file total modified lines out of all churned lines for the project [15]. If there is only one sub-project, then this value will be 100%.

8. Aggregate Characteristics

Aggregate candidate ACs come from the above ACs and provide a deeper understanding about an alert. Prior models measure age in days [8, 15]. Instead, we measure age as the number of revisions between two revisions. Using revisions is still a measure of time, but also provides a measure of work.

- **Alerts for Revision:** Number of alerts identified on or before an alert's open revision. As the revision numbers increase the number of alerts for that revision will increase as well.
- **Open Alerts for Revision:** Number of open alerts identified on or before an alert's open revision. Unlike the alerts for revision AC, the number of open alerts for revision will change due to alert closures.
- **Alert Lifetime:** The age of the alert [8] in number of revision. For a closed alert, the alert lifetime is the difference between the close and open revisions. Otherwise, the lifetime is the difference between the last revision in the study and the open revision.
- **File Age:** The age of the file [15] in revisions. For a deleted file, the file age is the difference between the deletion and creation revision. Otherwise, the file age is the difference between the last revision in the study and the file creation revision.
- **Alerts in Method:** The number of alerts in the method [3, 4] containing an alert across all revisions. The number of alerts in a method provides a relative measure of potential fault-proneness for a method.
- **Alerts in File:** The number of alerts in the file [3, 4, 15] containing an alert across all revisions. The number of alerts in a file provides a relative measure of fault-proneness for a file.
- **Alerts in Package:** The number of alerts in the package [3, 4] containing an alert across all revisions. The number of alerts in a package provides a relative measure of fault-proneness for a package.
- **Alerts in Project:** The number of alerts in the project [15] containing an alert across all revisions. The number of alerts in a project provides a relative measure of fault-proneness for a project.
- **File Staleness:** The amount of time between the last change of a file containing an alert and the last revision analyzed of the project [15].
- **Package Staleness:** The amount of time between the last change of a package containing an alert and the last revision analyzed of the project [15].
- **Project Staleness:** The amount of time between the last change of a project containing an alert and the last revision analyzed of the project [15].

9. AC Measurement Framework

The AC measurement framework is used to generate ACs for use in building FP mitigation models. For any project, the above ACs are gathered from static analysis tools, metrics tools, and the source code repository. By providing a measurement framework, AC generation can be shared between different FP mitigation models allowing for more direct comparison of FP mitigation models.

10. References

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis Defect Warnings On Production Software," *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, CA, USA, June 13-14, 2007, pp. 1-8.

[2] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for Bugs in All the Right Places," *International Symposium on Software Testing and Analysis*, 2006, pp. 61-71.

[3] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques, to appear," *2nd International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, October 9-10, 2008.

[4] S. S. Heckman, "Adaptively Ranking Alerts Generated from Automated Static Analysis," in *ACM Crossroads*. vol. 14, no. 1, 2007, pp. 16-20.

[5] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*: Prentice Hall, 1996.
[6] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 24-28, 2004, pp. 132-136.
[7] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.
[8] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, May 19-20, 2007, p. 27.
[9] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?," *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 3-7, 2007, pp. 45-54.
[10] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004, pp. 83-93.
[11] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," *10th International Static Analysis Symposium*, San Diego, California, 2003, pp. 295-315.
[12] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *28th International Conference on Software Engineering*, Shanghai, China, May 20-28, 2006, pp. 452-461.
[13] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," *International Symposium on Software Testing and Analysis*, 2004, pp. 86-96.
[14] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. Boston: McGraw Hill, 2005.
[15] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," *30th International Conference on Software Engineering*, Leipzig, Germany, May 10-18, 2008, pp. 341-350.
[16] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466-480, 2005.
[17] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Amsterdam: Morgan Kaufmann, 2005.