Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach

Joseph Ronald Ruthruff
University of Nebraska-Lincoln, ruthruff@cse.unl.edu

John Penix
Google Inc.

J. David Morgenthaler
Google Inc.

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08.
Digital Object Identifier: 10.1145/1368088.1368135

Distinguished Paper

Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach

Joseph R. Ruthruff*, John Penix†, J. David Morgenthaler†,
Sebastian Elbaum*, and Gregg Rothermel*

*University of Nebraska–Lincoln
Lincoln, NE, U.S.A.
{ruthruff, elbaum, grother}@cse.unl.edu

†Google Inc.
Mountain View, CA, U.S.A.
{jpenix, jdm}@google.com

ABSTRACT

Static analysis tools report software defects that may or may not be detected by other verification methods. Two challenges complicating the adoption of these tools are spurious false positive warnings and legitimate warnings that are not acted on. This paper reports automated support to help address these challenges using logistic regression models that predict the foregoing types of warnings from signals in the warnings and implicated code. Because examining many potential signaling factors in large software development settings can be expensive, we use a screening methodology to quickly discard factors with low predictive power and cost-effectively build predictive models. Our empirical evaluation indicates that these models can achieve high accuracy in predicting accurate and actionable static analysis warnings, and suggests that the models are competitive with alternative models built without screening.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability, Statistical methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis; G.3 [**Probability and Statistics**]: Correlation and regression analysis

General Terms

Experimentation, Reliability

Keywords

static analysis tools, screening, logistic regression analysis, experimental program analysis, software quality

1. INTRODUCTION

Static analysis tools detect software defects by analyzing a system without actually executing it. These tools utilize information from fixed program representations such as source code, generated or compiled code, and abstractions or models of the system. Even relatively simple analyses, such as detecting pointer dereferences after null checks, can find many defects in real software [5, 9].

There are well-known challenges regarding the use of static analysis tools. One challenge involves the accuracy of reported warnings. Because the software under analysis is not executed, static analysis tools must speculate on what the actual program behavior will be. They often over-estimate possible program behaviors, leading to spurious warnings ("false positives") that do not correspond to true defects. For example, Kremenek et al. [13] report that at least 30% of the warnings reported by sophisticated tools are false positives. At Google, we have observed that tools can be more accurate for certain types of warnings. Our experience with FindBugs [1] showed that focusing on selected, high priority warnings resulted in a 17% false positive rate [3].

A second challenge receiving less attention is that warnings are not always acted on by developers even if they reveal true defects. In the same study at Google, only 55% of the legitimate FindBugs warnings were acted on by developers after being entered into a bug tracking system [3]. Reasons for defects being ignored include warnings implicating obsolete code, "trivial" defects with no impact on the user, and real defects requiring significant effort to fix with little perceived benefit. Low criticality warnings such as "style" warnings, for example, can be unlikely to result in fixes.

We are investigating automated tools to help address both of these challenges by identifying *legitimate warnings that will be acted on by developers*, reducing the effort required to triage tens of thousands of warnings that can be reported in enterprise-wide settings. Our reasons for focusing on legitimate warnings are clear. We further focus on warnings that will be acted on by developers—not because ignored warnings are unimportant, but because we seek to maximize the return on investment from using static analysis tools. The core elements of our approach are statistical models generating binary classifications of static analysis warnings. One unique aspect of these models is that they are built using *screening*, an incremental statistical process to quickly discard factors with low predictive power and avoid the capture of expensive data.

Sampling from a base of tens of thousands of static analysis warnings from Google, we have built models that predict whether FindBugs warnings are false positives and, if they reveal real defects, whether these defects would be acted on by developers ("actionable warnings") or ignored despite their legitimacy ("trivial warnings"). The generated models were over 85% accurate in predicting false positives, and over 70% accurate in identifying actionable warnings, in a case study performed at Google. Both represent a notable improvement over previous practices at Google for FindBugs, where 24% of triaged warning reports had been false positives and 56% had been acted on. The results from this study also indicate that screening can yield large savings in the time required to generate models, while sacrificing little predictive power.

The primary contributions of this paper are:

1. a methodology for screening factors related to static analysis warnings that provides an economical means of generating accurate predictive models;
2. a set of measurable factors for static analysis warnings and programs, and the results of screening these factors in this context; and
3. logistic regression models for predicting both false positives and actionable warnings.

After introducing FindBugs and logistic regression techniques, we present the set of factors and describe their screening. We then present an empirical evaluation of the accuracy of the resulting logistic regression models using three baseline controls.

2. BACKGROUND

2.1 FindBugs at Google

FindBugs [9] is an open-source static analysis tool for Java programs. The tool analyzes Java bytecode to issue reports for 286 bug patterns [1]. These patterns are organized into seven categories: Bad Practice (questionable coding practices), Correctness (suspected defects), Internationalization, Malicious Code Vulnerability, Multithreaded Correctness, Performance, and Dodgy (confusing or anomalous code). FindBugs assigns reported warnings a priority of "High," "Medium," or "Low" based on each warning's estimated severity and the tool's confidence in its accuracy.

Evaluations of FindBugs on commercial software suggest that bug patterns have different false positives rates, and are of varying importance to developers when the warnings are accurate [3]. For example, warnings about null dereferences are generally susceptible to false positives because a tool cannot statically determine infeasible control flow paths between null assignments and dereferences. However, in some cases, a null dereference is assured because there is a single path between a null assignment and a dereference. This latter class of warnings is assigned a higher priority by FindBugs and has a low false positive rate. Regarding fixes, developers generally find null dereference defects important. But when the null dereference occurs while constructing an Exception, the observed fault is that the wrong exception is thrown. This may impact only exception logs and may not be given a high fix priority.

At Google, we have deployed FindBugs using an enterprise-wide service model [15]; this involves automatically collecting, building, and analyzing various portions of the code base on a repeated basis. The resulting warnings are triaged by a dedicated team and, when appropriate, reported to developers. We performed a cost/benefits analysis identifying this as a cost-effective approach for determining sufficiently interesting defects to report to developers. This analysis also indicated that, on average, eight minutes were required to manually triage each static analysis warning.

The scale of enterprise-wide deployment is challenging in a large, fast moving organization. Because static analysis tools perform "whole program" analysis, the need to re-analyze large parts of a software system could arise from small changes. Failure to do so may result in missed warnings or, if the tool is conservative, an increase in false positives. Running FindBugs frequently on a large, evolving code base generates numerous warnings. Over the nine-month period described in this paper, many tens of thousands of *unique* warnings were generated by FindBugs.[1] Because large numbers of new warnings are likely to be a common occurrence in large development settings, we believe that it is desirable that support for these tools, such as methodologies to build predictive models for incoming warnings, be adaptable, flexible, and scalable.

2.2 Logistic Regression Analysis

Logistic regression analysis is a type of categorical data analysis for predicting dependent variable values that follow binomial distributions [8]. Logistic regression models predict the value of a dependent variable y for a unit i based on the sum of a series of coefficients multiplied by the levels of particular independent (i.e., design) variables. To mathematically characterize this, let $X_i = (X_{i1}, \ldots, X_{in})$ be a vector of independent variable values for unit i. Let $\beta = \beta_0, \beta_1, \ldots, \beta_n$ be coefficients that are estimated by fitting the model to an existing "model building" data set, where β_0 is termed the model "intercept." Logistic regression models predict the binary value Y_i for i by calculating the probability that $Y_i = 1$ given X_i. This probability is:

$$P(Y_i|X_i) = \frac{e^{\beta_0+\beta_1 X_{i1}+\beta_2 X_{i2}+\cdots+\beta_n X_{in}}}{1+e^{\beta_0+\beta_1 X_{i1}+\beta_2 X_{i2}+\cdots+\beta_n X_{in}}} \qquad (1)$$

For details on logistic regression and how Equation 1 is derived, we refer readers elsewhere [8, 10].

3. LOGISTIC REGRESSION MODELS

Triaging static analysis warnings in an organization as large as Google is an expensive task. We aim to build statistical models that classify incoming static analysis warnings to reduce the cost of this process. We initially considered a number of potential factors to build these models, which are presented in Section 3.1. To increase the adaptability and scalability of our models and the process used to create them, we systematically screen these factors according to the procedures in Section 3.2 to incrementally filter out factors with low predictive power.

3.1 Logistic Regression Model Factors

3.1.1 Selecting Factors

We chose our factors and associated metrics by drawing from our own experiences with static analysis warnings at Google, as well as the experiences of other researchers who have built regression models in other software engineering domains. We draw primarily from three related areas of application. First, we capture some code complexity metrics as does the work of Nagappan et al. [16] for predicting post-release defects. However, in this work, we decided to focus on complexity measures that are more light-weight and can be very quickly computed on large code bases.

Similarly, we utilize work by Bell et al. [4, 18, 19] in building regression models for predicting fault counts within individual files in software releases. Their work considered factors relating to the size of the program, the recent change history of files, the recent fault history of files, files' age since the previous release, the programming language, and the release number. Because we consider only Java code here, and because we are interested in defects for code irrespective of release, we do not consider programming languages and releases as factors. Many of our selections are related to the first four aforementioned factors, however.

Finally, Kremenek et al. [13] consider factors related to the fault history of files in an adaptive probabilistic technique for ranking warnings, in addition to the files and directories themselves that contain warnings. We do not consider factors directly related to file and directory locality in this work, though many of our factors estimate other attributes of files and directories. Kremenek et

[1]The warnings are unique in that we used the FindBugs instance-hash method to track warnings across file changes [21]. This serves to identify previously-reported and duplicate warnings; however, it would not identify cases where a single defect triggers multiple unique warnings, which we currently must detect manually.

Factor	Description
FindBugs warning descriptors	
Pattern	Bug pattern of warning
Category	Category of warning
Priority	FindBugs warning priority
Google warning descriptors	
BugRank	Google metric of warning's priority
BugRank Range	Category (range) of warning's BugRank
File characteristics	
File age	Number of days that file has existed
File extension	Extension of Java file
History of warnings in code	
File warnings	Number of warnings reported for file
File staleness	Days since warning report for file
Package staleness	Days since warning report for package
Project warnings	Number of warnings reported for project
Project staleness	Days since warning report for project
Source code factors	
Depth	How far down (%) in file is warning
File length	Number of lines of code in file
Indentation	Spaces indenting warned line
Churn factors: files, packages, and projects (6×3 *factors*)	
Added	Number of lines added
Changed	Number of lines changed
Deleted	Number of lines deleted
Growth	Number of lines of growth
Total	Total number of lines changed
Percentage	Percentage of lines changed

Table 1: Factors for static analysis warnings and programs.

al. [13] also discuss the use of churn factors similar to our own; in our work, we investigate churn factors in detail.

3.1.2 Original Factors

We selected 33 factors to incorporate into the experimental screening methodology for generating our required models (Table 1), described in more detail below.

FindBugs Warning Descriptors. For a given static analysis warning, the FindBugs warning descriptor factors are taken directly from the FindBugs tool. As discussed in Section 2.1, there were 286 possible warning patterns, seven categories, and three priorities.

Google Warning Descriptors. To help triage warnings, we developed a prioritization scheme that we call BugRank. BugRank ranks warnings based on previous triage and bug-fix statistics. Specifically, it uses the empirical history of how often warnings on a given pattern have been false positive and how often they have been acted on by developers as a probability of each event happening. This information is blended with a default probability based on the FindBugs category and priority. As warnings are triaged and acted on, the weighting of the observed data increases. In this paper, we used the BugRank associated with warnings at the end of May 2007.

BugRank is represented as a range from 0-100, where 0 indicates a likely defect that will be fixed, and 100 indicates an inconsequential or spurious warning. BugRank categories are ranges of BugRank values, where 0-10 is Very High, 11-20 is High, 21-50 is Medium, 51-90 is Low and 91-100 is Very Low.

File Characteristics. We consider two factors related to characteristics of files generating static analysis warnings. Our first factor was inspired by Bell et al. [4, 18]: the age of files. Because we do not use software releases here, we consider the number of days since the file's creation. Our second factor is related to Google development practices: file extension. We use FindBugs to analyze both standard and generated Java code, and many generated files can be differentiated from standard Java files by file extension.

History of Warnings in Code. We consider five factors designed to capture the history of static analysis warnings at three granularities of code: files, Java packages, and projects. The types of factors we consider for these granularities are "number of warnings" and "staleness." The two number of warnings factors concern the total number of warnings that have been historically reported for the files and projects in question at the time of screening. This factor is in part inspired by those in related work [4, 13, 18] concerning the history of prior defects. The staleness factors capture the number of days that have passed since the most recent warning was first reported for the file, package, or project at the time of screening.

Source Code Factors. We considered three simple factors that aim to provide insight into the code generating static analysis warnings. The first considers the depth of the implicated code in the file, in terms of percentage of lines of code. Our second factor analyzes the program to measure the number of spaces indenting the implicated line of code. The notion behind this factor is that deeply nested code is more complex than code in outer scopes. The third factor considers file size in terms of total lines of code, which was another factor considered in related work [4, 18].

Our source code factors are inspired by code complexity; however, they are purposefully not as sophisticated as traditional complexity measures. In addition to the inconsistent predictive power displayed in other work by some traditional complexity factors [16], due to the frequency at which static analysis warnings can be reported and our desire to make predictions about incoming warnings quickly, we wanted to determine whether a few inexpensive factors could provide sufficient predictive power in models.

Churn Factors. Code churn is a general measure for the amount of changing code, and how the code has changed. We consider 18 churn factors at three levels of granularity; these three levels are the file, package (including sub-packages), and project levels.

For each level of granularity, we consider six churn factors. The "added" factor considers the number of lines that have been added to the file, packages, and project in the three months prior to the date on which the warning was reported. We selected three months because we did not want too short a window (e.g., one week) that might not consider enough churn; this is especially relevant for the file granularity, as many files are not modified frequently. However, too long a window (e.g., one year) might be too noisy to have predictive power. The "changed" factor considers the number of changed (edited) lines in the prior three months for the three levels of churn granularity. The "deleted" factor considers the number of lines that have been deleted. The "growth" factor considers the code's growth in terms of lines of code; this can be either a positive or negative number. "Total" considers the total number of churned lines, and "percentage" considers the percentage of churned lines.

3.2 Experimental Screening Process

Screening experiments are designed to quickly yet systematically narrow down large groups of independent variables, or independent variable levels (treatments), into smaller subsets to further investigate. They are often used as formative, exploratory work to focus the direction of research, and in settings where large numbers of treatment combinations and interactions are of interest.

We have used the experimental program analysis framework [20] to create a screening methodology for selecting, from many potentially expensive factors, a subset that could be used as independent variables in logistic regression models for static analysis warnings and programs. The goal of the screening methodology is to converge on independent variables for logistic regression models that accurately predict false positives and actionable static analysis warnings in a cost-effective manner.

One reason to seek a cost-effective approach is that it may be desirable to re-build predictive regression models at various points in time such as when a significant number of new warnings have been reported, the codebase of interest has experienced substantial change, or it is of interest to investigate new factors. The frequency in which it may be desirable to re-build models due to such changes would likely vary in different software development settings. However, we expect that many settings would use existing models to predict static analysis warnings for at least a moderate length of time before updated models are built.

In this work, we consider a screening methodology with up to four stages that attempts to identify at least six predictive factors for a predictive model. We selected four stages to accommodate ranges of 5%, 25%, 50%, and 100% of the total warnings, the reasons and goals of which are presented in the upcoming discussion. We chose a cut-off of six factors because we wanted to ensure that we were left with at least some factors for model building. Six factors is also close to the number of factors considered in the models developed by Nagappan et al. [16] and Bell et al. [4, 18].

The first stage of the screening methodology considers 5% of static analysis warnings and source code. The goal of this stage is to consider a small subset of warnings and eliminate factors that appear to have little of the predictive power needed to build accurate models. After gathering data for this small subset of warnings, we build a logistic regression model from the resulting data and perform an Analysis of Deviance for generalized linear models [6] to individually evaluate each factor.

Because statistical models are designed to closely model the data on which they were built, Analysis of Deviance considers how well each factor in the model reduces the fitted model's deviance from the fitted data. As a logistic regression model, we expect our model to fit data with a binomial dispersion, so Chi-squared tests are appropriate for generating test statistics to evaluate each factor's reduction in deviance [6]. We used these tests to evaluate each factor at each screening stage. In this first stage, we eliminated factors with effect sizes so small that the test statistics' p-values were greater than 0.80. These factors were judged to contribute little to the model's goodness-of-fit to the model building data.

If performing this elimination left six or fewer factors, we halted screening, and gathered data using the remaining 95% of warnings according to the remaining factors. Otherwise, we considered an additional 20% of the static analysis warnings in a second stage, bringing the total number of considered warnings to 25%. (We chose 25% so that one-quarter of the total warnings would be considered after this stage.) An Analysis of Deviance is then used to eliminate factors whose test statistics' p-values were greater than 0.50. Because this stage considers a larger amount of data than previous stages, we chose an elimination criterion that was more strict than in the first stage, but was still lenient enough to allow factors to recover when more data is available in subsequent screening.

The third stage of our screening methodology considers the next 25% of warnings, for a total of half of all warnings. The elimination criterion here is more strict: p-values greater than 0.20. This was chosen because, by this point, a sizable amount of data has been gathered with which to make informed judgments.

If more than six factors remain, then these factors are used to gather data for the last 50% of the data. After this fourth and final stage, factors with non-significant p-values greater than 0.10 are eliminated. The logistic regression model is then fit in R using the remaining factors as independent variables.

3.3 Building Models From Screening Factors

We used the screening methodology described in Section 3.2 to filter out factors listed in Section 3.1 with insufficient predictive power. For a set of static analysis warnings to use for collecting the necessary data during screening, we chose a sample of 1,652 static analysis warnings reported by FindBugs for Java code at Google. (This sample is also the subject of our case study; see Section 4.) We present the resulting model for predicting false positive warnings, built using the R statistical package [2], followed by the models for predicting actionable warnings.

3.3.1 Model for Predicting False Positives

Examining just 5% of the static analysis warnings in the first stage of the screening experiment eliminated 15 of the 33 factors initially considered. The 15 eliminated factors were (1–6) all six project churn factors, (7–10) the growth and total churn factors for both files and packages, (11) the churn percentage factor for files, (12) the churn factor for changed lines in files, (13–14) the staleness factors for projects and files, and (15) the file extension. This stage took 29 minutes and 57 seconds, with 62% of the time required to screen the six project churn factors, 32% of the time required to screen the six package churn factors, and 6% of the time for the remaining 21 factors.

Five factors were eliminated in the second stage of screening: (16–17) the churn factors for lines deleted in files and packages, (18) the churn factor for lines added in packages, (19) the churn percentage in packages, and (20) the number of warnings in the project. Because an additional 20% of the sampled warnings were considered, this stage required 45 minutes and 26 seconds to complete, even though only 18 factors were screened. This amount of time was largely due to the four remaining package churn factors, which required 84% of this time to screen and happened to be eliminated in this stage.

The factors for (21) the file's age and (22) number of warnings in the file were eliminated in the third stage, which took nine minutes and seven seconds to complete. In the fourth and final stage, the factors for (23) the FindBugs category and (24) the file churn in terms of changed lines were eliminated. Although only 11 factors remained in this stage, it took 14 minutes and 53 seconds to complete since the last 50% of the sampled warnings were processed.

The nine factors selected by screening, as well as the intercept (i.e., the β_0 coefficient) of the regression model, are summarized in Table 2. The significance of each factor on the model's deviance from the fitted data, as well as the coefficients of the fitted model for each factor, are shown. The categorical variables such as the BugRank range have coefficients for all but one level, which is folded into the intercept.[2] In Tables 2–4, the labels for the priority and BugRank range factors are as follows: VL stands for "Very Low," L for "Low," M for "Medium," and VH for "Very High."

The adjusted R^2 for this model, which describes the amount of variation in the dependent variable that the model captures, is 0.3548. Although this might seem low, it is actually quite reasonable given the complexity of what the models are predicting. It also indicates that there may be room for improvement in the form of additional factors that capture more variation. However, we also

[2] R alphabetically folds the first level of categorical factors into the intercept, which is "High" for priority and BugRank ranges.

Factor	$P(>\chi)$	Coefficients
Intercept		-2.29064
FindBugs pattern	< 0.01	*(not shown)*
FindBugs priority	< 0.01	M: 1.67002, L: 0.83176
Package staleness	0.07	-0.00227
BugRank	< 0.01	0.23924
BugRank range	< 0.01	VL: -9.02230, M: -8.24013, L: -15.12205, VH: 2.83218
File churn: Added	0.03	0.00088
Code: Warning depth	0.04	-0.00678
Code: Indentation	0.10	-0.02704
Code: File length	0.03	-0.00032

Table 2: Factors selected through screening for false positive prediction models. Coefficients for pattern are not shown.

note that there is some inherit variation in the setting we are observing (e.g., triaging engineers may not always agree on whether a warning identifies a real problem), which is likely to keep R^2 low.

The coefficients indicate how each factor affects the predictions made by the model. (For the value predicted in Equation 1, values close to 0.0 correspond to false positive predictions, while values close to 1.0 correspond to true defects.) For example, the coefficient of the package staleness factor is negative. As more time passes since the previous static analysis warning in the package or sub-packages, the model is more likely to predict that the warning is a false positive. In contrast, the coefficient for the churn factor for lines of code added to files is positive; thus, the model is more likely to predict that the warning is legitimate as the number of lines recently added to files increases.

3.3.2 Models for Actionable Warnings

We consider two types of models to predict actionable warnings. Our first model is built using only those warnings identified as true defects. For building this model, 12 factors were eliminated after the first screening stage: (1–6) all six project churn factors, (7–10) the growth and total churn factors for both files and packages, (11) file extension, and (12) BugRank. The second stage saw the elimination of (13) file length complexity and (14) the number of warnings in the project. The third stage eliminated (15) line indentation and (16) the churn factor for changed lines in packages. Four factors were eliminated in the fourth and final stage: (17–18) the staleness of the packages and projects, (19) the churn factor for changed lines in files, and (20) the FindBugs priority.

The 13 factors remaining after screening are summarized in Table 3. For the category factor, C stands for "Correctness," I for "Internationalization," MC stands for "Malicious Code," MTC for "Multi-Threaded Correctness," P for "Performance," and S for "Style." The adjusted R^2 for this model is 0.2477.

Our second model is designed to predict actionable defects from all warnings (i.e., both false positives and legitimate warnings). Screening eliminated 18 factors: the six factors for project churn; the changed lines, growth, and total lines churn factors for files and packages; the package churn factor for added lines; indentation complexity; file length; package and project staleness; and file extension. The 15 factors remaining after screening are summarized in Table 4. The adjusted R^2 for this model is 0.2611.

3.3.3 Discussion

It is interesting to observe trends in the factors that were consistently selected and eliminated during screening. For example, the project churn factors were consistently eliminated after the first screening stage. Because many Google projects undergo a large amount of churn, and because project churn operates at a high granularity, this factor may be too noisy to be useful in regression models for static analysis warnings, at least for three-month windows of time. On the other hand, other factors such as bug pattern and BugRank range were consistently selected by screening.

We also saw trends within particular models. For example, the actionable warnings model for true defects included the added, deleted, and percentage churn factors at both the file and package level, and excluded the changed, growth, and total factors for the same levels. All of these trends suggest areas for future work regarding additional factors that, we believe, could further improve the precision of our models.

There could be benefit in investigating certain factors further, and in considering new factors. For example, we considered a three-month window for the churn factors, but larger or smaller windows could be considered. Also, some factors, such as the number of warnings and staleness factors, are measured at the point of screening rather than the time of the warning report. We measured at

Factor	$P(>\chi)$	Coefficients
Intercept		-1.69250
FindBugs pattern	< 0.01	*(not shown)*
FindBugs category	< 0.01	C: 0.13550, MTC: 0.29493, I: 13.41806, P: -0.44291, MC: -13.52201, S: 0.29889
File age	< 0.01	-0.00096
Warnings in file	< 0.01	0.08125
File staleness	< 0.01	0.00764
BugRank range	< 0.01	VL: 16.44913, L: -0.35007, M: -1.19761, VH: 1.16869
File churn: Added	< 0.01	-0.00078
File churn: Deleted	< 0.01	0.00083
File churn: Perc.	< 0.01	0.00608
Dir churn: Added	0.02	0.00009
Dir churn: Deleted	0.01	-0.00013
Dir churn: Perc.	< 0.01	0.01024
Code: Warning depth	0.03	-0.00623

Table 3: Factors selected for actionable prediction models considering only true defects. Coefficients for pattern are not shown.

Factor	$P(>\chi)$	Coefficients
Intercept		-24.76222
Pattern	< 0.01	*(not shown)*
Priority	< 0.01	L: 1.17921, M: 1.87402
Category	< 0.01	C: 1.58666, S: -0.21193
Warnings in file	< 0.01	0.067965
File age	< 0.01	-0.00107
File staleness	< 0.01	0.00588
Project staleness	0.01	0.00488
BugRank	0.03	0.08894
BugRank range	< 0.01	VL: 5.10045, M: -4.55509 L: -6.98454, VH: 2.66199
File churn: Added	< 0.01	-0.00025
File churn: Deleted	0.01	0.00027
File churn: Perc.	< 0.01	0.00408
Dir. churn: Perc.	< 0.01	0.00894
Code: Warning depth	0.01	-0.00739

Table 4: Factors selected for actionable prediction models considering all warnings. Coefficients for pattern are not shown.

this point primarily with actionable warnings in mind; such a fluid factor allows us to measure whether warnings are actionable at particular points in time. (As time passes, for example, code may become obsolete and older warnings may become less actionable. On the other hand, a sequence of recent warning reports may spur interest in a previously neglected file.) However, there is merit to other approaches, which future work could investigate.

Some of our factors are collinear, and models fit with such factors can suffer from reduced precision due to increases in standard errors. We investigated collinear factors in this work because, at this early stage in the work, we wished to consider many factors to determine those that are most preferable and should be pursued further. Finally, some factors were captured through non-continuous variables. For example, "bug pattern" is a categorical variable of the nominal type. This type of variable requires some manipulation before inclusion in a regression model. Throughout this study, we used the default mechanism provided by R to mark and encode this type of variable for usage in the regression models.

4. CASE STUDY

In this section we evaluate the models generated in Section 3 for predicting false positive and actionable static analysis warnings. In addition to evaluating the accuracy of these models, we also evaluate the time taken to collect the data according to the screened factors, and to build the models in a statistical package. This evaluation took place as a case study using a large set of FindBugs static analysis warnings at Google. We compared the models generated from our screening methodology against three "controls."

4.1 Design

4.1.1 Setting and Warning Samples

Our case study was performed using static analysis warnings reported against Google's Java codebase. The data set consists of 1,652 unique warnings selected from a population of tens of thousands of warnings seen over a nine-month period from August 31, 2006 to May 31, 2007. The warnings in the data set were manually examined and classified as either false positives or true defects by two Google engineers from the triage team described in Section 2.1. The warnings deemed true defects were incorporated into bug reports and assigned to developers judged responsible for the code.

These engineers generally focused on higher-priority warnings because finding important bugs was the primary goal [3]. Sometimes, BugRank was used to prioritize warnings to be examined. Other times, warnings were prioritized by project. Warning patterns with only one or two total warnings were generally examined. Our sample of warnings included 157 of the 286 FindBugs bug patterns and all seven FindBugs categories. The distribution by FindBugs priority was: 38% High, 60% Medium, and 2% Low. The distribution by BugRank Range was: 27% Very High, 32% High, 28% Medium, 12% Low, and less than 1% Very Low priority.

We stopped considering warnings on May 31, 2007 to allow over three months for the most recently reported warnings to be resolved by developers; this step was taken to help prevent a warning's report date from influencing whether it was acted on within the context of this study. The mean time to resolve defects reported by warnings during this period was 34 days, with a median of 10 days.

4.1.2 Independent Variables

The independent variable evaluated in this case study was the model used to make predictions about warnings. A conjoined factor is the methodology used to generate the models, as the methodology determines the data available for model building.

Screening models. As treatment variables, we considered the three models for classifying warnings that were built from our screening methodology; we refer to these models as `Screening`. For control variables, we compared each treatment model against three baseline models and their accompanying design methodologies.

All-Data models. As our first control, we attempted to collect data for every factor listed in Section 3.1 for every sampled warning. This baseline helps us assess the gains in cost, as well as any loss in precision, from screening factors. However, it is generally ill-advised to build regression models using every factor without some form of filtering. While many factors may be useful for predicting false positives and actionable warnings, others will have little predictive power due to little correlation to the sampled warnings. Variable selection methods are often used to select the most predictive factors that best estimate the dependent variable, or to eliminate the least predictive factors [10].[3] We selected backward elimination for this first baseline, which we label `All-Data`.

Some factors were too expensive to collect data for even within this study's sample of warnings. Specifically, the project churn metrics take an inordinate amount of time to collect due to the large amount of code in some projects. Because collecting data for these metrics would be a significant drain on the limited number of available shared resources used by many Google engineers, we decided not to consider project churn in `All-Data`. (However, we note that these factors were quickly eliminated in screening due to their apparent noise and lack of predictive power.)

BOW models. As controls, we wished to consider regression models used by other researchers. Although we are not aware of published models built for predicting characteristics of static analysis warnings, they have been used in other software engineering domains. Our next two controls are based on the work of Bell et al. [4, 18], which we considered when designing our own factors.

Table 5 compares factors from [4, 18] with related factors from Section 3.1. Our second control builds models using the factors represented in the right-most column in Table 5. To build these models, we collect data according to these six factors for the sampled static analysis warnings. We refer to these models and their data collection methodology as the `BOW` models.

Our third control is an extension of the `BOW` models. While the factors in `BOW` are based on years of research and development, they were considered in a different context: predicting fault counts in released software systems. Because we are considering models for predicting static analysis warnings, the `BOW` models may lack the necessary context to provide accurate predictions in this setting. To provide this context and set up a potentially more fair comparison between `BOW` and `Screening`, we extended the `BOW` models in our third control by adding the "bug pattern" and "priority" factors. We selected these because they are two standard factors describing static analysis warnings that are provided directly by FindBugs, and they are fine-grained, which may reduce the noise in the factors' data as compared to more coarse-grained factors like "category." We refer to these models as the `BOW+` models.

4.1.3 Dependent Variables

We formulated two constructs to evaluate our models: the accuracy of the logistic regression models' predictions, and the cost of building the models. We measure the time taken to build each model as a dependent variable. This variable considers the sum of

[3]Such "post-mortem" elimination is actually the opposite of our screening approach, which seeks to eliminate useless factors as early as possible, rather than after all data has been collected.

Factor	Ostrand et al. 2004 [18]	Bell et al. 2006 [4]	This work (Section 3.1)
Code size	Lines of code	Lines of code	Lines of code in file
Change History	File changes (new, changed, unchanged)	Recent change history	Churned lines (file: added, changed, deleted)
File age	Number of releases file has appeared	File age in a given month	File age (# days file has existed)
# Faults	Number of faults in previous release	Recent history of faults	Number of previous warnings in file

Table 5: Relating factors from Bell et al. to similar factors in this work.

the time required to gather the needed data from the warnings and Java programs, and to generate the models in R.

Our dependent variable for measuring accuracy compares the predicted status of each warning to its known, triaged status, and divides the number of correctly predicted warnings by the total number of warnings. For false-positive-predicting models, we consider the predicted and known false positive status of each warning. For models predicting actionable warnings, we consider whether each legitimate warning led to developer actions resulting in a defect fix, and compare this status with the models' predictions. In Section 4.2.2, we examine the incorrect predictions made by the models, and compare the R^2 values of the `All-Data` models to the `Screening` models.

4.1.4 Procedures

It is difficult to measure the accuracy of regression models in an unbiased manner, and there are many designs for measuring model precision. One common strategy is the "resubstitution" strategy, where the models are used to classify the same data on which they were fit. This design allows models to be evaluated using all available data, which would generally increase model precision, and supports the evaluation of more representative models because the models used in practice are generally built from the largest possible sample of data. A disadvantage of this design is that it suffers from bias, as it generally overestimates the probability of correct classifications; however, Johnson [10] says that this bias is relatively low for large data sets. Given these trade-offs, we use resubstitution as our first design, and consider a second design to provide an additional view of our study's data.

Our second design uses holdout data, which is another common strategy for evaluating regression models [17]. In holdout designs, a percentage of the data set is withheld as a "validation" set, while the remainder is used as a "model building" set. This approach does not suffer from the bias of resubstitution; however, because it does not maximize the amount of model-building data, intuitively, it may underestimate model precision. There is also the issue of how much data to withhold for validation. Withholding a small amount of data allows the model to be built with more information; however, a validation set that is too small increases the likelihood of anomalous prediction measurements, and makes it more difficult to draw meaningful conclusions about the accuracy of the models in general cases. To find balance in the presence of these trade-offs, we consider three ratios of model building data to validation data in our holdout designs: 70% model building data to 30% validation data, 80% model building to 20% validation, and 90% model building to 10% validation. Furthermore, to limit the possibility of anomalous results from one particular data set, we randomly select three sets for each of our data ratios, and separately build and evaluate models for each set of data within each ratio.

To collect the timing information regarding the cost of building the prediction models for each independent variable, we measure the time taken to gather the data according to the selected factors (or, in the case of screening, to screen the factors according to the procedures in Section 3.2), and the time taken to build the models in R. For the `Screening` models, the data gathering step includes the periodic statistical analysis performed to analyze the significance of the measured factors on the false positive or actionable warning effect being measured. For the `All-Data` models, this latter effort includes the time required to perform the backward elimination to filter out factors with low predictive power.

Because the accuracy of the `All-Data` predictions are measured without the project churn factors, we consider both the time required to build these models without the project churn data, and the estimated time with project churn. Similarly, because we want to compare `Screening` with `All-Data` to gauge the cost savings from screening factors, we measure the time required to build the `Screening` models both with and without project churn. (Project churn is not considered in the `BOW` models, so we measure time only once for these models.)

4.1.5 Threats to Validity

We consider three types of threats to the validity of this case study's results [22]: external validity, internal validity, and construct validity. We took various measures to mitigate our study's external validity threats. In terms of our sample of warnings, we sampled over 1,600 warnings over a nine-month period to help ensure that our study was representative of the setting at Google. This sample considered 157 FindBugs bug patterns, and all seven FindBugs categories. FindBugs is also one of the more sophisticated and widely adopted static analysis tools available for Java.

We used the warnings examined by two engineers to help control for variation in triaging decisions among similar warnings because we wanted to isolate the effect of the models' predictive power in this study. In practice, we would expect greater variation in these decisions due to triaging by additional engineers. Finally, different results may be experienced in different software development settings, with alternative factors for building logistic regression models, and with different static analysis tools other than FindBugs. This threat can be addressed only by further empirical studies.

The bias in resubstitution designs is an internal validity threat to the results from this methodology, though this concern is mitigated due to this study's large sample of warnings. Another internal validity threat to the holdout designs is the particular validation set used, as different validation sets may contribute to different results. This latter threat is stronger for the smaller validation sets. For these reasons, we considered three ratios of model-building to validation data, and three randomly selected assignments of data within each ratio. Also, using two validation approaches reduces the threats of drawing conclusions from just one approach.

Because engineers may classify warnings differently, to help ensure consistent classifications, we selected only warnings examined by one of two engineers; however, this came at the cost of external validity. The statistical generation of these logistic regression models assumes that the occurrence of false positive and actionable warnings are binomially distributed. If these assumptions are not met in our data, then the power and precision of the models may decrease. Finally, although FindBugs offers support to track duplicate warnings, a single defect could be responsible for multiple unique warnings reported by the tool. We allowed for this possible bias because this scenario is likely to also face real practitioners.

Model Type	**Data Gathering**			**Model Building**		
	False Pos.	*Actionable: Defects*	*Actionable: All*	*False Pos.*	*Actionable: Defects*	*Actionable: All*
`Screening`	0:01:39:23 (0:00:42:36)	0:06:48:10 (0:03:42:55)	0:06:57:11 (0:03:55:02)	< 00:01	< 00:01	< 00:01
`All-Data`	5:05:50:06 (0:04:21:01)			03:38	00:22	00:39
`BOW`	0:00:39:00			< 00:01	< 00:01	< 00:01
`BOW+`	0:00:39:05			< 00:01	< 00:01	< 00:01

Table 6: Cost of building the four types of models. For "data gathering," the top row for `Screening` and `All-Data` is the time with project churn; the second result is without project churn. The format of Data Gathering times is Days:Hours:Minutes:Seconds. The format of Model Building times is Minutes:Seconds.

While we believe our constructs are reasonable, threats to construct validity primarily derive from the legitimacy of the comparisons of our screening-based models to our controls. With regard to the `All-Data` models, as described earlier, it was not feasible for us to gather the entirety of the data for every static analysis warning for every factor; in fact, this scalability issue was a motivation for using screening in the first place. Excluding some factors from `All-Data` was the most feasible way to design this control to evaluate the cost-effectiveness of screening.

With respect to the `BOW` models, these models were designed using factors similar, but not identical, to those used in related work [4, 18]. Furthermore, the BOW models are used in a different context than the one they were created for. We attempted to mitigate this concern by also evaluating the `BOW+` model, which considers static analysis factors to give the models context with respect to the environment in which they are used in this study.

4.2 Results and Discussion

We first discuss results pertaining to the cost of building the models. We then discuss the accuracy of the models' predictions and the cost considerations required for this accuracy.

4.2.1 Cost

Table 6 summarizes the time taken to generate the logistic regression models, which consists of both data gathering and model fitting. Data gathering is performed once according to the models' data collection methodologies. The model building times shown are the times taken to build the models using the entire sample of data, as was done in Section 3.3. Recall from this section that `Screening` selected more factors for the models predicting actionable warnings than did the model predicting false positives, accounting for the increased data gathering time for these two models.

It can take a large amount of time to collect the data needed to build the `All-Data` regression models when expensive metrics like project churn are considered. In this case, `Screening` was effective at reducing the cost of collecting data according to these factors by filtering out factors such as project churn and file age that, according to the screening, did not have the predictive power to be considered further. By doing so, `Screening` took only a fraction of the time for data gathering as did `All-Data`. Even when the expensive project churn metrics were not considered, the savings experienced through screening was notable; for example, screening 28 factors required only 42 minutes for the false-positive-predicting models.

The `BOW` models required a reasonable 39 minutes to collect data according to their six to eight fixed factors. It was more expensive to build the `Screening` models than the `BOW` models—not surprising considering the number of factors initially considered at the beginning of screening, and the expense of many of those factors. However, when the most expensive project churn metrics were not considered in `Screening`, the time required to build the regression models, particularly the models predicting false positives, was closer to that for the `BOW` models. Of course, `Screening` and `BOW` reflect two different types of scenarios. The `BOW` models reflect a situation where there are fixed factors, known beforehand, to be investigated. Screening factors, on the other hand, will generally be more expensive than collecting data for the foregoing scenario, depending on the cost of the factors being screened.

Finally, it is important to note that generating models is unlikely to be a one-time cost. As new tools are added, existing tools are updated with new bug pattern detectors, and factors for model building are added or updated. It can become necessary to gather data on at least a subset of new warnings if updated models are to be built. Our results indicate that `Screening` could be a cost-effective means of collecting data in these types of scenarios.

4.2.2 Prediction Accuracy

Tables 7 and 8 summarize the accuracy of the logistic regression models in predicting false positives and actionable static analysis warnings. As can be seen, the `Screening` models were generally the most accurate in predicting false positives and actionable warnings based on true defects (Table 7 and the top of Table 8). The results indicate that `Screening` did not lose predictive power as compared to the `All-Data` models for these types of predictions, even doing better than the `All-Data` models. The R^2 values for the `All-Data` models also indicate that the `Screening` models captured approximately the same amount of variation as the `All-Data` models. The R^2 values for the `All-Data` models predicting false positives and actionable warnings from true defects were 0.3586 and 0.2812, respectively, compared to 0.3548 and 0.2477 for the `Screening` models.

These results were surprising. We had expected the `All-Data` models to be the most accurate due to their ability to leverage the most available data when building models. It may be that incrementally screening factors is preferable to performing stepwise elimination procedures following data collection. It may also be that collecting *too much* data at once in this type of setting is not desirable because it inherently leaves a greater amount of noise in the data that procedures such as stepwise elimination must try to reduce. Performing this elimination in a systematic manner while data is being collected, as does `Screening`, may be the preferable method for selecting design variables for logistic regression models for static analysis warnings.

`Screening` was not as precise, as compared to `All-Data`, for predicting actionable warnings based on both false positives and true defects (bottom of Table 8). However, as discussed in Section 4.2.1, this minor loss of precision came with a substantial reduction of cost in terms of building the models. Also, the R^2 values indicate that the `Screening` models actually captured slightly more variation (0.2611) than the `All-Data` models (0.2511).

Model Type	Resubstitution	Holdout Data		
		70/30	*80/20*	*90/10*
`Screening`	85.29%	87.48%	87.09%	86.54%
`All-Data`	85.71%	83.48%	84.74%	85.47%
`BOW`	76.51%	77.96%	78.73%	79.32%
`BOW+`	84.62%	82.24%	83.81%	83.06%

Table 7: Predicting false positive warnings. Holdout data shows the average precision of the models from the three observations.

Model Type	Resubstitution	Holdout Data		
		70/30	*80/20*	*90/10*
True Defects				
`Screening`	77.32%	71.82%	71.68%	71.95%
`All-Data`	71.37%	71.42%	69.95%	70.97%
`BOW`	60.19%	61.30%	63.47%	62.74%
`BOW+`	70.90%	67.04%	67.41%	69.79%
All Warnings				
`Screening`	77.42%	72.02%	71.36%	71.94%
`All-Data`	73.73%	72.90%	75.26%	75.68%
`BOW`	62.23%	59.35%	60.77%	61.12%
`BOW+`	73.91%	67.76%	69.50%	69.26%

Table 8: Predicting actionable warnings.

The `Screening` models were also more precise than the `BOW` and `BOW+` models as shown in Tables 7 and 8. These two models performed relatively well for having been built using factors from a different software engineering domain. While they were not as precise as the `Screening` models, they were somewhat less expensive to build. Although our area of interest was one where we have many factors to investigate for static analysis models, this finding suggests that practitioners who are not interested in investigating many factors, and wish instead to use a fixed number of known factors, may find some success if those factors are well selected.

In terms of the deployment of FindBugs at Google, the `Screening` models represent an improvement with respect to previous practices. We observed that 24% of examined warnings in our sample, for all FindBugs bug patterns, were false positives. For actionable warnings, only 56% of the static analysis warnings indicating true defects were acted on by developers; the percentage of actionable warnings drops to 42% when considering all static analysis warnings—not just those indicating true defects. However, our results indicate that the deployment of the logistic regression models could decrease the number of false positives examined by engineers from 24% to as low as 12.5%, and could ensure that over 70% of the warnings examined by developers would be acted upon.

Had these models been used within our sample of 1,652 warnings, triaging engineers would have examined hundreds of fewer spurious and non-actionable warnings—not an insignificant savings when the cost of triaging is estimated at eight minutes per warning (as discussed in Section 2.1). However, this comes at a cost due to the incorrect predictions made by the models.

Not shown in Tables 7 and 8 is a breakdown of where the models went wrong when incorrectly predicting warnings. For the false positive models in the 70/30 holdout design in Table 7, 33% of the incorrectly predicted warnings were predicted to be false positive when the warnings were actually legitimate, while 67% of incorrectly predicted warnings were predicted to be legitimate when the warnings were false positive. For the actionable models based on true defects in the same design (top of Table 8), 40.3% of the incorrectly predicted warnings were predicted to be trivial warnings, but were actually acted on by developers, while 59.7% of the warnings were predicted to be actionable, but were not acted on.

Though not by intent, the models tended to be conservative. Because there were more false positives predicted to be legitimate warnings, and more non-actionable warnings predicted to be actionable, triaging engineers would see more of the types of warnings they wished to avoid (e.g., false positives) than important warnings that should not be missed (e.g., warnings identifying true defects). Generally speaking, the costs of examining more false positives versus missing legitimate warnings (and, similarly, non-actionable versus actionable warnings) needs to be evaluated in the context of the software being analyzed with static analysis, and the development organization as a whole. In future work, improvement in the areas where the models go wrong in terms of incorrectly predicted warnings is likely to come through more sophisticated static analysis tools, additional factors with improved predictive power, and a larger sample of warnings on which to build models.

5. RELATED WORK

To our knowledge, we are the first to look closely at predicting developer response to static analysis warnings. There are several threads of related work, however, that use statistical models to filter or prioritize static analysis warnings. Heckman [7] has recently proposed the use of adaptive models that utilize feedback from developers to rank warnings in order to identify false positives. These rankings are derived from models using factors whose coefficients change after developer feedback, and were effective in favorably ranking and identifying false positive warnings [7].

Kremenek and Engler developed a ranking algorithm (z-ranking) to prioritize warnings for a single warning type (i.e., a single "checker") [14]. The algorithm is based on their observation that clusters of warnings are usually either all false positives due to tool limitations, or all true defects due to developer confusion. Z-ranking was effective at prioritizing legitimate warnings above false positives [14]. A generalization of this work produced an adaptive ranking scheme called Feedback Rank [13]. This work used code locality (function, file, and directory) to identify clusters of false positives and legitimate warnings. The adaptive aspect of the algorithm updates warning priorities as nearby warnings are classified.

Because we triaged priority warnings across an entire codebase rather than focusing on all the warnings for particular files, our data set appears to be too sparse to directly support correlation by file locality: 85% of legitimate warnings and 80% of false positives appear as the only warning from a file. We do have a number of file, package, and project factors that would be the same for warnings reported at the same time, for the same file. If these factors account for some of the file locality correlation observed by Kremenek et al., our models should correlate warnings for different files that have similar values, as measured by factor metrics. We also include the general notion of warning clustering by including warning counts for files; this factor was selected by screening for both of our models predicting actionable defects. While our logistic regression models are not adaptive, our BugRank factor is because it incorporates observed, evolving data regarding false positives and resolved warnings.

Kim and Ernst [11, 12] use fixed information mined from software change histories to estimate the importance of warnings. They use this information to improve the default priorities assigned by the tools. For our data set, we have actual "fix data" from our bug tracking system that should be more accurate than data mining change histories. However, our data is then limited to the warnings filed as defects against developers. In theory, we could expand our data set to also include warnings that disappeared during changes. However, the cost of mining version histories would be similar to the cost of computing code churn, which is prohibitively expensive in our setting. There were also notable differences in our observations that indicate the need for further study in this area. For example, "equals used to compare unrelated types" is one of our most frequently fixed warnings. However, Kim and Ernst observe long delays in fixing these warning patterns and rank them low [11]. This is an interesting area for further study.

The models presented in this paper operate by classifying each warning rather than ranking all warnings. It would be interesting to see how classification and ranking models compare in practice.

6. CONCLUSIONS

The return on investment for static analysis tools like FindBugs is limited by the cost of checking whether reported warnings correspond to faults and, as important, of assessing whether such warnings are worth further effort. The magnitude of this limitation is evident both in prior work and in our study of 1,652 triaged warnings, where approximately 24% of warnings were false positives, and only 56% identifying true defects were acted on by developers.

In this paper we aimed to maximize static analysis tools' return on investment by developing models that predict whether reported warnings constitute actionable faults. Given that our setting is characterized by the possible need to regularly re-build and adjust models, and by constraints on large-scale metric collection, it was also important to devise an approach that would enable efficient model building. The proposed screening approach for model building accomplishes this by quickly discarding metrics with low predictive power—avoiding their collection throughout a large code base.

In our empirical study we found that the predictive power of the models built on screened data was, in general, at least as good as that of the models utilizing the whole warning data sets, while incurring significantly less data collection effort. The screening-based models were able to accurately predict false positive warnings over 85% of the time on average, and actionable warnings over 70% of the time. Furthermore, the screening models consistently performed better than existing predictive models that we adapted from a slightly different software engineering context. As expected, however, such improvement required additional data collection.

Based on this experience, and more generally, we conjecture that screening will be particularly useful at early stages of model building, when there are an abundance of factors considered to have an effect on the target dependent variable, and especially appealing in the presence of factors whose associated metrics are expensive to collect. This work also indicates that regression models may be effective in settings involving static analysis warnings, and shows promise for future work in this area.

We have worked to deploy these models across our development practices at Google in order to reduce the number of needlessly examined warnings by triaging engineers. When generalized to the tens of thousands of warnings that have been reported in our software development environment, the savings from using these models could be substantial.

7. ACKNOWLEDGMENTS

Joseph Ruthruff was an intern at Google Inc. when this work was performed. YuQian Zhou, Simon Quellen Field, and Larry Zhou contributed to the static analysis infrastructure at Google. William Pugh greatly assisted in our use of FindBugs and suggested factors to consider for our regression models.

8. REFERENCES

[1] FindBugs. http://findbugs.sourceforge.net/.

[2] The R project for statistical computing. http://r-project.org/.

[3] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proc. 7th ACM Workshop on Prog. Analysis for Softw. Tools and Eng.*, pages 168–179, 2007.

[4] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Looking for bugs in all the right places. In *Proc. ACM Int'l Symp. on Softw. Testing and Analysis*, pages 61–71, 2006.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symp. on OS Principles*, 2001.

[6] T. J. Hastie and D. Pregibon. *Statistical Models in S*. Wadsworth & Brooks/Cole, 1992.

[7] S. S. Heckman. Adaptively ranking alerts generated from automated static analysis. *ACM Crossroads*, 14(1), 2007.

[8] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression*. John Wiley & Sons, 2nd ed., 2000.

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to Proc. OOPSLA*, pages 132–136, 2004.

[10] D. E. Johnson. *Applied Multivariate Methods for Data Analysis*. Duxbury Press, 1998.

[11] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *Proc. Int'l Workshop on Mining Softw. Repositories*, 2007.

[12] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. 6th Joint ESEC/SIGSOFT Foundations of Softw. Eng.*, pages 45–54, 2007.

[13] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *Proc. 12th ACM Int'l Symp. Foundations of Softw. Eng.*, pages 83–93, 2004.

[14] T. Kremenek and D. Engler. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. 10th Static Analysis Symp.*, 2003.

[15] L. Z. Markosian, O. O'Malley, J. Penix, and W. Brew. Hosted services for advanced V&V technologies: An approach to achieving adoption without the woes of usage. In *Proc. ICSE Workshop on Adoption-Centric Softw. Eng.*, 2003.

[16] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th Int'l Conf. on Softw. Eng.*, pages 452–461, 2006.

[17] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. Irwin, 4th edition, 1996.

[18] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proc. ACM SIGSOFT Int'l Symp. on Softw. Testing and Analysis*, pages 86–96, 2004.

[19] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Automating algorithms for the identification of fault-prone files. In *Proc. ACM SIGSOFT Int'l Symp. on Softw. Testing and Analysis*, pages 219–227, 2007.

[20] J. R. Ruthruff, S. Elbaum, and G. Rothermel. Experimental program analysis: A new program analysis paradigm. In *Proc. ACM SIGSOFT Int'l Symp. on Softw. Testing and Analysis*, pages 49–59, 2006.

[21] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 133–136. ACM Press, 2006.

[22] C. Wohlin, P. Runeson, M. Host, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.