Warning-Introducing Commits vs Bug-Introducing Commits: A tool, statistical models, and a preliminary user study

Louis-Philippe Querel
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
l_querel@encs.concordia.ca

Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
peter.rigby@concordia.ca

***Abstract*—This paper partially replicates prior works on building historical commits [1], commit risk modeling [2], and a comparison of statistical bug models and static bug finders [3].**

We examine 8 Maven-based projects with an average lifespan of 5.8 years. To historically build these projects across a total of 45k commits, we develop a series of techniques, such as flexibly selecting the version of a library that is closest to the commit date. We are able to build a per project average of 78.4% of all commits, a doubling in buildability compared to prior work. We also develop a git blame strategy to assign warnings even when a commit does not build.

We run JLint and FindBugs and create a logistic regression model to predict if a commit that introduces a warning has higher odds of introducing a bug. The static bug finders model accounted for only 13% of the deviance, while the statistical bug model accounted for 19.5%. We had expected static bug finder warnings to improve the predictive power of models of bug introducing changes, but we clearly attained a negative result.

To understand this negative result, we perform a preliminary user study of developers who introduced new warnings in 37 projects. We found that while warnings might not predict bugs, 53% and 21% of warnings in Findbugs and Jlint respectively are useful. We also study whether just-in-time warnings presentation on each commit impacted usefulness. We find that the later a warning is shown to a developer, the less useful it is perceived to be (a median of 11.5 days versus 23 days for non useful warnings).

Based on our findings, we modify the existing COMMITGURU interface to add new warnings to the specific line in a changed file. The empirical study data [4] and WARNINGSGURU tool [5] are publicly available.

***Index Terms*—bug detection, static bug finder, statistical bug prediction, user study**

I. INTRODUCTION

Static analysis finds faults by examining the code, data-flow, and control-flow for problematic patterns. To make static analysis tractable in practice static bug finders, such as FindBugs and JLint, make simplifications and abstractions that lead to a large number of warnings many of which can be false positives, including trivial and unlikely warnings [6], [7]. The advantage of static bug finders is that the warnings are specific, *e.g.,* a null pointer on a specific source code line. The disadvantage is the overwhelming number of reported warnings and a disconnect between field defects and warnings [8].

In contrast, statistical bug models use historically mined development information, such as the number of lines changed and the developer's experience, to indicate risky, *i.e.* potentially defective modules, files, and bug-introduction commits [9], [10], [2], [11]. The advantage of statistical bug models is that they provide reasonable predictions of field defects in commits and files [11]. The disadvantage is that the prediction is not fine-grained, *e.g.,* an entire file or commit is flagged as potentially bug introducing because the file has been changed recently.

In this paper, we partially replicate work on statistical bug models using change measures [2] and compare them with static bug finders performing a replicating with a differing, negative result [3]. Before we can create these models, we first need to build historical commits and replicate work by Tufano *et al.* [1]. We describe our research question below.

RQ1, Builds and Warnings. What proportion of historical commits can be built and analyzed by static bug finders?

To extract the static bug finder warnings from the history of a project each commit needs to be buildable. Tufano *et al.* [1] attempt to build 279k commits across 100 projects using Maven. They find that for the average project only 38.13% of the commits can be built. The biggest barrier to building historical commits is resolving dependencies.

Replication Result: We examine over 45k commits across eight large projects with an average history of 5.8 years. Our technique uses dependencies outside Maven Central and also uses older versions of libraries that are closer in date to the the time of the commit. We are able to build an average 78.4% of commits on each project. While we examine fewer projects than Tufano *et al.*, on a per project basis we are able to proportionally build twice as many commits.

We then run static bug finders on the builds. We use git blame to trace warnings back to their original commit, which may not necessarily build. In this way, we are able to assign all the warnings to their warning-introducing commit regardless of the build status.

RQ2, Change Measures. Using change measures, how well can we predict bug-introducing changes?

We create a baseline statistical bug model using historically mined development information to identify bug-introducing commits. We need this baseline model to be able to later control for the the predictive power of the typical measures used in the research literature, such as churn and developer expertise [2], [11].

Replication Result: We create a logistic regression of the commits that introduce bugs and change measures. The model has reasonable fit with the deviance explained at 19.5%. This is a partial replication of Kamei *et al.* [2] on a new set of projects.

RQ3, Static bug finder warnings. Do the warnings present in a commit predict bug-introducing changes?

A commit that contains a new warning should be at higher risk to introduce a bug in the system than a commit with no warnings. We use the presence of warnings to statistically model the odds of introducing a bug. We break warnings into categories, including JLint vs FindBugs warnings, security warnings, and new vs existing warnings.

Negative Result: The model explains only 13% of the deviance. When we combine the change measures and warnings into a single model we can explain 22% of the deviance. Even when compared with a simple baseline model that does not reach the sophistication of the state-of-the-art, the addition of warnings on commits does not improve the predictive power. This finding is a negative replication of prior work that examined warnings on a small number of releases and found that change measures and static bug finder warnings had near equivalent power in identifying field bugs [3].

Although static bug finder warnings do *not* substantially improve the predictive power of traditional statistical models, static bug finders are not designed to catch all types of bugs, for example, they were never intended to catch a usability bug [6]. To better understand this negative result, we assess the *perceived* usefulness of static bug finder warnings in a user study and answer the following research questions.

RQ4, Perceived usefulness. How useful do developers perceive warnings to be?

We create dynamic survey requests that displayed the commit and the new warning to the author of the commit. We sent surveys to 179 developers from randomly sampled commits and received perceived usefulness ratings on 81 warnings. We purposefully left the definition of usefulness at the discretion of the developer.

Result: 53% and 21% of warnings in Findbugs and Jlint are perceived as useful by developers. Although warnings may be poor predictors of bug-introducing changes, they often provide useful information to the developer.

RQ5, Recency. Does the time lag from the introduction of a warning to when a developer sees it influence the perceived usefulness of the warning?

Some teams review warnings periodically, *e.g.,* at a security review, while others examine the warnings with each commit. We want to determine the impact of recency on the perceived usefulness of warnings (just-in-time warnings). We randomly vary the delay in sending the warning survey to a developer.

TABLE I
PROJECTS UNDER STUDY

Project	Years	Commits	Success	Failure	Build Success
Commons-lang	10.2	3314	3123	191	94.2 %
Hadoop	5.5	14458	9360	5098	64.7 %
Ignite	1.7	4368	3606	762	82.6 %
Kylin	2.4	5749	5084	665	88.4 %
Phoenix	3.0	1892	1818	74	96.1 %
Ranger	3.0	1913	961	952	50.1 %
Tika	9.9	3345	3166	179	94.7 %
Wicket	10.6	10910	6164	4746	56.5 %
Average:	5.8	5744	4160	1583	78.4 %
Total:	–	45949	33282	12667	72.4 %

Result: Useful warnings were a median of 11.5 days old, while non-useful warnings were a median 23 days old. Recently introduced warnings are perceived as more useful than older warnings.

The practical outcome of our work is the WARNINGSGURU tool.[1] The tool uses a statistical model to flag risky commits and then indicates which lines contain warnings. We also flag the warnings as new to the current commit or show the commit that originally introduced an existing warning. The empirical study data [4] and WARNINGSGURU tool [5] are publicly available.

This paper is structured as follows. In Section II, we describe WARNINGSGURU's processing pipeline, the projects we analyze, and the descriptive statistics on the buildability of commits and the number of new warnings per commit. In Section III, we introduce the change history measures and warnings counts that will be included in our models to predict bug-introducing commits. In Section IV, we conduct a user study to understand how developers perceive the usefulness of warnings. In Section V, we discuss how our design of the WARNINGSGURU interface was driven by our statistical models and user study results. In Sections VI, VII, and VIII we respectively discuss threats to validity, related work, and conclude the paper.

II. WARNINGSGURU PIPELINE, PROJECTS, AND DATA

WARNINGSGURU integrates source version control, build tools, and static bug finders to build and analyze thousands of historical commits. Using WARNINGSGURU we identify which commit a warning originates from and which warnings are new to a commit. We also develop a technique to assign warnings even when a commit does not build.

A. Processing Pipeline

Figure II-A illustrates our processing pipeline and we discuss the stages in detail below:

1. Git Commits. WARNINGSGURU obtains the list of commits from COMMITGURU [10]. This includes historical commits and new commits as COMMITGURU updates the repositories. WARNINGSGURU analyses commits incrementally from the newest to oldest. Once a commit has been analyzed the results are stored in a database.

[1]We build upon the publicly available preliminary version of the tool that was presented in the tool demo track of FSE [12]

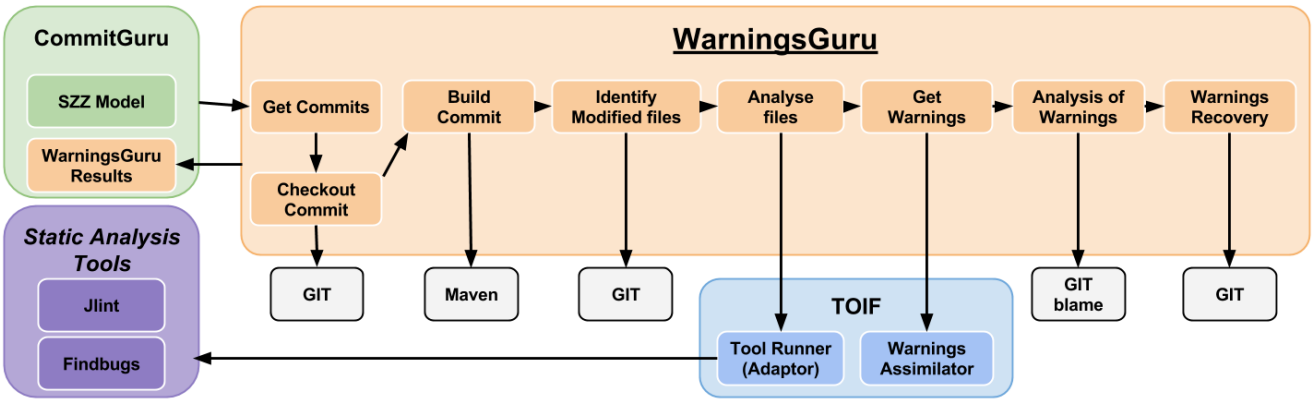

Fig. 1. WARNINGSGURU architecture

2. Building Commits. WARNINGSGURU depends on the Apache Maven POM configuration file [13] and builds the commit. Tufano *et al.* [1] found that on average only 38.13% of a project's historical commits could be built. We developed strategies to increase the buildability of the history of projects. We observed that some were not specifying in their POM files the version of Java which they were targeting to be used as part of their build. While newer versions of the Java Development Kit (JDK) should be able to build code which was meant for older versions of Java, the lack of targeting resulted in incompatibilities which caused builds to fail. WARNINGSGURU implements a mechanism to override the version of the JDK used for building based on the date of a commit to build older commits which do not support the newer versions of Java. Since we override the JDK, WARNINGSGURU also provides a mechanism to override the version of Maven to ensures that a compatible version of the build tool is employed to perform the build.

The parameters used by the projects can also evolve with subsequent releases. Some of the projects have their build configured to fail where a test on the application is unsuccessful. As we are only building to obtain the build artifacts, we are not interested in the results of the tests which would significantly reduce the performance of WARNINGSGURU. We therefore disable test build failures.

The memory requirements for a project were not specified in the Maven files. To improve buildability, additional memory is allocated as part of the execution which WARNINGSGURU performs by default. This reduces the risk that the build would fail due to insufficient memory.

While Maven provides mechanisms to manage the dependencies, it is still possible that a build fails due to a missing project configured dependency. To mitigate this issue we have added additional repositories to our Maven instance that include dependencies that are not present in Maven Central. Furthermore, projects would often use a 'SNAPSHOT' dependency of a intermediate version of library that no longer exists. In these cases, we replaced the intermediate versions with a version that exists and is closer to the date of the commit.

Warnings. WARNINGSGURU runs Findbugs and JLint and extracts the list of warnings associated with the modified files. TOIF [14] augments these results with CWE [15] and SFP [14] which are a warning type and security-related type classification, respectively.

Historical Warning Reassignment. Static bug finders assign warnings to a line of code but do not determine when a warning was introduced. We use Git blame to identify when the line last changed and reassign warnings on a line to the commit in which the line was last changed. Since Git blame does not require a commit to build, we can retroactively assign warnings to lines added even when a commit does not build.

B. Studied Projects

We analysed the following projects: Commons-lang, Hadoop, Ignite, Kylin, Phoenix, Tika, Ranger and Wicket. Each projects has between 1.9 and 14k commits over multiple years, see Table I. The project are written primarily in Java and use the Apache Maven build tool to manage their build configuration and are available on GitHub. We briefly describe each project. Commons-lang is a library which provides additional utilities to the core Java classes. Hadoop is a distributed computing and storage platform. Ignite is an in-memory computing platform. Kylin is a distributed analytical engine which interacts with solutions such as Hadoop. Phoenix is a library that provides SQL support for non-SQL databases. Tika is a tool which extracts metadata from files which can be used for indexing. Ranger is a monitoring and security utility for Hadoop. Wicket is a web framework to build Java server side based services.

C. Result for Builds and Warnings

RQ1: What proportion of historical commits can be built and analyzed by static bug finders?

Tufano *et al.* [1] studied the buildability of 100 projects over 249k versions and found that an average of 38.13% of commits could be built per project. As described above, we

TABLE II
BREAKDOWN OF COMMITS WITH WARNINGS. WE ARE ABLE TO RE-ASSIGN WARNINGS TO FAILED BUILDS. INTERESTINGLY, THE NUMBER OF WARNINGS ON PASSING VS FAILING BUILDS IS SIMILAR.

	Commits	Percentage
Total	45949	-
With warnings	26898	58.5%
With new warnings	5881	12.8%
Total successful builds	33282	72.4%
Successful with warnings	19553	58.7%
Successful with new warnings	4387	13.2%
Total failed builds	12667	27.6%
Failed with warnings	7345	58.0%
Failed with new warnings	1494	11.8%

varied the version of the JDK and other libraries to match the date closest to the commit and included repositories outside Maven Central. Using these techniques, on a smaller sample of 8 projects, we are able to proportionally build twice as many commits with a per project average of 78.4% of the commits. The full breakdown is in Table I.

Prior works that combine static analysis with statistical bug models include only a small number of project snapshots and do not provide a tool or replication package [3], [16]. For example, Rahman *et al.* [3] study between 5 and 8 release for five open source projects for a total of 34 releases. They state that the effort to build and run JLint, FindBugs, and PMD on the 34 versions took six person-months of effort. Tang *et al.* [16] studied 3 and 5 releases of 2 projects for a total of 8 releases. Nanda *et al.* [17] created a private tool which ran static analysis tools on commits, but did not build current or historical commits.

Table II shows the breakdown of warnings and builds. We extracted the warnings from 45,949 commits an increase of 1,351 times as many versions compared to Rahman *et al.* [3]. 58.5% or 26,898 commits contained warnings and 12.8% or 5,881 introduced at least one warning. In total, we identified 940k distinct warnings. As noted by prior works, building historical versions of projects is difficult and 27.6% or 12,667 of the commits did not build. However, using our Git blame strategy we were able to reassign 256k warnings to their originating commit. Without this approach, 28.2% of the total warnings would be attributed to the wrong commit. By processing all the commits of a project, WARNINGSGURU gives precise and timely information about when a warning *first* appeared and avoids the complex accounting schemes used by prior works to compare static bug finders with statistical bug models.

We build over 45k builds across eight large projects with an average history of 5.8 years. Our technique uses dependencies outside Maven Central and also uses older versions of libraries that are closer in date to the the time of the commit. We are able to build a per project average of 78.4% of the commits, a doubling in buildability compared to prior work. We assign each static bug finder warning to the originating commit regardless of whether it builds. We find that 58.5% of commits contain warnings and 12.8% introduce new warnings.

III. STATISTICAL MODELS

Statistical bug models have been used to predict the occurrences of bugs in projects. They use historically mined development information to indicate risky, *i.e.* potentially defective, files or commits, *e.g.,* [2], [11].

We investigate the use of warnings as a change measure for statistical bug prediction models. Warnings have previously been used by Rahman *et al.* [3] and Tang *et al.* [16], but they were using a small number of snapshots of projects. Since they use release snapshots they cannot determine the exact change that introduced a warning. WARNINGSGURU assigns warnings to commits allowing us to assess how well warnings predict bug introducing commits.

We systematically built a series of models to assess the predictive power of warnings in predicting bug-introducing changes and compare them with COMMITGURU's change measures model. The models are logistic regressions with the binary response measure being whether a commit introduced a bug, *i.e.* IsBugIntroducting. These bug-introducing commits are found by first identifying the commit that fixes a bug based on keywords in their commit message, *e.g.,* "fixing bug #14934." Using Git blame, the fixing commit is traced back to the commit that last changed the fixed lines [18], [10].

We compare three models: a traditional model including the COMMITGURU change measures, a static analysis warnings model, and the combined COMMITGURU and warnings model. We describe the measures and model creation below.

1) Building the COMMITGURU *model:* The predictors or independent variables are measured are at the commit level and are divided into churn and developer measures. Since these measures have been widely used in COMMITGURU, we only briefly summarize them here and refer the reader to Kamei *et al.* [2] for further details. The final churn measures are the number of modified directories, lines added, lines removed, and the LOCs in the files before the change. The final developer measures are the number of developers that have modified the files in the change, the authoring developer's experience measured as the number of prior changes the developer has made, and the recent developer experience weighted by the number of changes made in each year. We also include the binary "fix" variable that indicates whether the commit is fixing an existing bug.

We run a Spearman correlation among the measures and keep the most parsimonious measure when the correlation is greater than 0.75. We excluded the number of modified files and entropy as they correlate at 0.94 and 0.89 respectively with the number of modified directories. We also excluded the number of changes to a file as it correlates at 0.78 with total line count before the file was modified. Due to the removal of the numbers subsystems measure, we also removed the developer's subsystem experience due to correlation of 0.74 with a developer's experience.

To represent our models, we use the R language notation. For example, the formula $y \sim a + b$ means that "the response y is modeled by explanatory variables a and b." As discussed above, the response variable is a binary variable indicating whether the commit introduced a bug, *i.e.* IsBugIntroducing. Following prior works [19], [20], we applied a log transformation to explanatory variables when they are right skewed with a long tail. To avoid taking the logarithm of zero, we add one to the variables. The project variable is added as factors to account for individual project differences. The final model is

$$\begin{aligned}\text{IsBugIntroducing} \sim\ & \log_2(\text{Number of directories} + 1) \\ & + \log_2(\text{Lines added} + 1) \\ & + \log_2(\text{Lines removed} + 1) \\ & + \log_2(\text{Lines before change} + 1) \\ & + fix \\ & + \log_2(\text{Number of developers} + 1) \\ & + \log_2(\text{Average time between changes} + 1) \\ & + \log_2(\text{Developer experience} + 1) \\ & + \log_2(\text{Recent developer experience} + 1) \\ & + \text{as.factor(Project)} \\ & \text{family=binomial()}\end{aligned}$$

2) Building the static bug finder model: Unlike previous works that work with a limited number of releases ([8], [3], [16]), we analyze the entire history of warnings on over 45k commits. Instead of indicating whether or not a commit has a new warning, we use counts of warning types. We measure the number of new warnings, and we are able to differentiate security warnings allowing us to determine when security warnings are introduced.

When building our warnings model we first include the number of new security warnings. We then include the number of new warnings, which is effectively the addition of interaction terms between the number of new security warnings + the number of new non-security warnings. We continue this process adding the the total number of security warnings and then the number of total warnings. In the model we also differentiate between the tool that found the warning, either FindBugs or JLint.

A final measure is whether the build failed. A failed build can demonstrate problems in the code and environment and may indicate significant problems with the commit.

The final R model for the Warnings model is

$$\begin{aligned}\text{IsBugIntroducing} \sim\ & \log_2(\text{NewSecurityWarnings} + 1) \\ & + \log_2(\text{SecurityWarnings} + 1) \\ & + \log_2(\text{NewFindbugsWarnings} + 1) \\ & + \log_2(\text{NewJlintWarnings} + 1) \\ & + \log_2(\text{FindbugsWarnings} + 1) \\ & + \log_2(\text{JlintWarnings} + 1) \\ & + \text{BuildFailed} \\ & + \text{as.factor(Project)} \\ & \text{family=binomial()}\end{aligned}$$

3) Combining the COMMITGURU *and static bug finder models:* We combine all the measures from previous models. We include the COMMITGURU measures first and the warning measures second because we want to determine the degree of redundancy of our warnings measures compared with the COMMITGURU measures. In the model, the measures are added in the order shown in Table III.

A. Results for COMMITGURU model

RQ2: Using change measures, how well can we predict bug introducing changes?

In the first column of Table III we see the results of the COMMITGURU change measure logistical regression bug model. We can see that the model explains a reasonable amount of the deviance, 19.5%. Since our predictors are skewed any non-categorical variable is transformed using log_2. We report the odds ratio for each predictor. However, since they are log_2 transformed they represent a twofold increase in the predictor. For example, a twofold increase in the number of directories touched, lines added, and lines before a change makes it 1.25, 1.44, and 1.15 times more likely for a bug to occur in a commit. Previous work has found that files with more churn tend to have more bugs, our findings confirm this suggesting that large changes introduce more bugs [21].

FIX which is the identification of a commit as being bug fixing, is binary. As a result a commit that is fixing an existing bug is 1.52 times more likely to introduce a bug. Our findings indicate that bug fixes likely touch fragile or complex code and lead to further bugs. This agrees with the findings that the number of past defects is a strong predictor of future defects [22].

The strongest negative predictor is the number of developers who touch a file, with a twofold increase in this predictor leading to a 26% decrease in the likelihood of the commit introducing a bug. This is surprising as this implies that the greater number of people modifying the files the fewer bugs. These results agree with Bird *et al.*'s [23] findings that showed that even geographically distributed developers

TABLE III
STATISTICAL MODELS PREDICTING BUG-INTRODUCING COMMITS

	COMMITGURU	Warnings	Combined
Num Directories	1.25 †		1.05 *
Lines added	1.44 †		1.39 †
Lines removed	1.03 †		1.02 *
Lines before change	1.15 †		1.08 †
FIX	1.52 †		1.51 †
Num of Devs on files	0.74 †		0.82 †
Avg time between changes	1.00		1.01
Developer Experience	1.05 †		1.04 †
Recent Dev. Experience	0.97 †		0.99
new security warnings		1.02	1.08
security warnings		0.89 †	0.93 †
new FindBugs warnings		1.43 †	1.10 *
new JLint warnings		1.58 †	1.14 †
FindBugs warnings		1.19 †	1.07 †
JLint warnings		1.35 †	1.22 †
build failed		2.00 †	1.75 †
Hadoop	0.56 †	0.57 †	0.42 †
Ignite	0.55 †	0.62 †	0.48 †
Kylin	0.84 *	1.01	0.85 *
Phoenix	0.38 †	0.28 †	0.26 †
Ranger	0.42 †	0.46 †	0.33 †
Tika	0.57 †	0.98	0.65 †
Wicket	0.82 †	0.89 *	0.69 †
Deviance Explained	19.5%	13.4%	22.0%
Residual	41195	44323	39928

Statistical significance: $\dagger p \ll 0.05$, $*p < 0.05$, otherwise $p \geq 0.05$

Table interpretation: We report the standard odds ratio and p value. With the exception of binary variables, all predictors are log_2 transformed. As a result, instead of a linear increase in odds, the odds ratio represents a doubling or twofold increase. For example, a twofold increase in the number of lines added increases the odds of a bug being introduced by 1.44 times. To compare the odds of introducing a bug for each project a reference project is arbitrarily choose, in this case Commons-lang. The odds ratio for each project shows the relative increase or decrease in the odds of introducing a bug.

did not introduce more bugs. Herbsleb and Mockus however identified that bug fixing was more prevalent with more geographical distributed developers [24]. Clearly future work into understanding the social interactions is required.

> We create a logistic regression of the commits that introduce bugs using COMMITGURU model that includes change measures. The model has reasonable fit with the deviance explained at 19.5%. Churn and whether a commit fixes a bug remain the strongest predictors of future bugs.

B. Result for static bug finder warnings model

RQ3: Do the warnings present in a commit predict bug introducing changes?

In the second column of Table III, we see the results of our static analysis warnings bug regression model. We can see that the model explains a smaller proportion of the deviance, 13.4%.

The strongest predictor of bug introduction is whether the build succeeded. A build failure doubles the likelihood of a bug being introduced. A twofold increase in the number of new warnings for a commit increases the likelihood of introducing a bug by 1.5 and 1.19 times for JLint and FindBugs respectively. Having more existing warnings in the code that is changed in a commit also increases the likelihood of introducing a bug. New security warnings were not statistically significant while the total number of security warnings actually reduced the likelihood of a bug. This security finding is likely related to the difficulty and rarity of actual security bugs, which as Camilo *et al.* [25] point out, make vulnerabilities difficult to predict statistically. Overall, our findings contradicts smaller studies that suggest that change measures and static analysis warnings have similar defect prediction potential [3], [16].

In the third column of Table III, we see the combination of the COMMITGURU change measures and static bug finder warning measures. The deviance explained is 22.0%, only 2.5 percentage points higher than the COMMITGURU change measures model. The measures are for the most part consistent in terms of direction and power with the largest drops in predictive power seen by the new warnings predictors. Our analysis suggests that the warning counts are largely redundant with COMMITGURU measures such as file size accounting for larger warning counts.

> The static bug finder warnings model explains 13.4% of the deviance. When we combine it with the COMMITGURU change measures the single explains 22% of the deviance. The increase over the COMMITGURU model is only 2.5 percentage points. The warnings counts are mostly redundant explaining similar phenomena. These negative results indicate that warnings are poor predictors of bug introducing changes.

IV. PERCEIVED USEFULNESS OF STATIC BUG FINDER WARNINGS

The statistical models in the previous section indicate that static bug finder warnings do not substantially improve the predictive power of COMMITGURU change measure statistical models. However, static bug finders are not designed to catch all types of bugs [6], for example, static bug finders were never intended to catch, a UI bug on the position of a button for improved usability. We conduct a preliminary study of how useful developers perceive static bug finder warnings to be.

A. Study Design and Methodology

To ensure a sufficient number of developers, we increased the number of projects from 8 to 37. We analysed the following Apache projects: Accumulo, Apex core, Apex malhar, Asterixdb, Beam, Brooklyn-server, Calcite, Canyenne, Cloudstack, Commons-lang, Commons-net, Commons-text, Crunch, Curator, Falcon, Hadoop, HBase, Ignite, Tamaya-Extension, Knox, Kylin, ManifoldCF, Oozie, OpenNLP, Phoenix, Ranger, Sentry, Storm, Tika, Tinkerpop, Twill, Wicket and Zeppelin.

For each project, we ran WARNINGSGURU. When a new warning was identified we sent the committing developer a

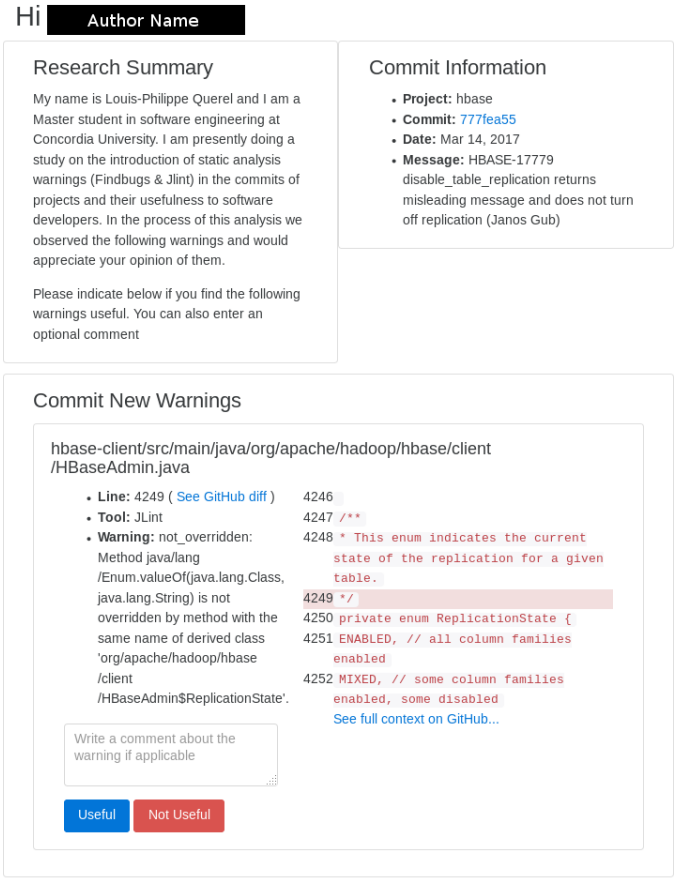

Fig. 2. The dynamically generated warnings survey page provides the developer with the warning and the context. The developer decides if the warning is useful.

survey asking if the new warning provided useful information. As shown in Figure IV-A, to assist the developers in their assessment, we provide the context of the warning by presenting the lines of code and its surrounding code as part of the warning.

We emailed the survey to 179 developers and obtained a response rate of 17.9%. In total, we obtained a usefulness rating for 81 warnings. While the number of total warnings is small for a study, as we will discuss, the results are statistically significant indicating a strong difference in what constitutes a useful warning. We make the tool to generate the survey from commits and static analysis warnings publicly available [4].

B. Results for perceived usefulness of warnings

RQ4: How useful do developers perceive warnings to be?

Table IV shows that only 34.6% of warnings are useful. This results concurs with prior work that many warnings are false positives [6]. Comparing the perceived usefulness of JLint vs Findbugs, we find that 52.9% of Findbugs warnings are useful as opposed to 21.3% for Jlint warnings (Table IV). To provide a statistical comparison, we use the Fisher Exact test and find $p < 0.05$ indicating that the difference is statistically significant and Findbugs warnings are perceived as more useful than JLint warnings. While future work may be necessary, as a linter, JLint likely points out potentially obvious syntax problems, while Findbugs provides warnings of more substance that have the perception of greater utility.

TABLE IV
DEVELOPERS' PERCEPTION OF STATIC BUG FINDER WARNING USEFULNESS

Tool	Warnings	Useful
JLint	47	21.3%
Findbugs	34	52.9%
Total	81	34.6%

> 52.9% and 21.3% of warnings in Findbugs and Jlint are perceived as useful by developers. Warnings may be poor predictors of bug introducing changes, but they often provide useful information to developers.

C. Results for Warning Recency

RQ5: Does the time lag from the introduction of a warning to when a developer sees it influence the perceived usefulness of the warning?

In our discussion with developers, we found that some teams examined warnings with each commit, while others periodically processed a large number of warnings before a release or during a security audit. To understand the impact of warning recency on perceived warning usefulness we conducted a preliminary study.

We sent requests to developers who introduced new warnings within a 45 day window. Some developers received the request immediately, while others received the survey request weeks after the commit. The time lag, developer, and warning were selected at random. This methodology allows us to determine the impact of warning recency on perceived usefulness.

We calculate the recency as a time delta in days between the introduction of a new warning by a developer and the time which we receive a response to our survey request, *i.e.* the time the developer examined the warning.

Recency results: Warnings that developers labeled as useful had a median time delta of 11.5 days between their introduction in the commit and the developer response. For warnings which were indicated to be non-useful by developers, there was a median of 23 days. Using a Wilcoxon test, we find the difference is statistically significant with $p = 0.018$. Figure 3 is a violin plot that shows the distribution of recency in days. There is a clear skew, indicating that the sooner a warning is seen the more likely that it will be perceived as useful. However, some old warnings do provide value with the oldest useful warning of 43 days.

These preliminary results have implication for tool design. Warnings should be seen by developer as soon as possible. Many teams already practice in this way adding hooks in

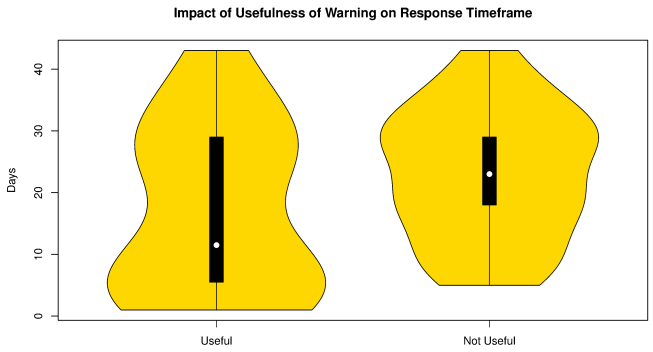

Fig. 3. The violin plots shows the distribution of perceived usefulness of a warning relative to recency delta in days. The dot represents the median of the dataset and the box plot shows the second and third quantiles. The sooner a warning is seen by a developers, the more useful the warning is perceived to be.

their continuous integration infrastructure to show warnings immediately at commit.

For researchers, this result opens an interesting area of future work. A preliminary hypothesis is that the developers forget the context of the commit as more time passes, which results in the developers identifying the warnings as not being useful. This would require additional research to assess if this could be an analog in software projects for the forgetting curve which is the study how people remember and recall information [26].

> Warnings that are perceived to be useful were more recent with a median time delta of 11.5 days, while non-useful warnings have a median of 23 days. Recently introduced warnings are perceived as more useful than older warnings.

V. WarningsGuru tool Interface

The design and integration WarningsGuru into the interface of CommitGuru is grounded in the results from the statistical models in Section III and the preliminary user study in Section IV. Figure 4, shows WarningsGuru integrated into CommitGuru.

A limitation of models that predict commit risk and bug introducing changes is that they do not provide a specific file or reason for flagging a commit. For example, in Figure 4, we see that CommitGuru has flagged commit *bea9e03aeb* as risky because there were many lines added and removed. This "rationale" does not provide a specific actionable guide to the developer that would allow him or her to ensure that this risky commit does not introduce a bug. We augment the statistical model by adding the warning counts to the CommitGuru. In Figure 4, we see that the commit has 2 new warnings and 15 total warnings.

Our user study shows that warnings are perceived to be more useful to developers when they are received in a timely manner. To provide timely actionable results, we show the number of warnings per file in the commit. When a file is selected, we show the line that the warning occurs on and a link to the source line on GitHub. Based on the results from our study on timelines, we also indicate to the developer which lines contain new warnings with a star ($\star$). For example, we see in the figure that on line 2658 in file *ConnectionQueryServiceImpl.java* the JLint warning that the "Comparison always produces the same result". The developer can also see that there are multiple warnings on line 1 that were introduced in an older commit *c5b80246*. The developer can click on the line or commit to see the change diff that introduced the warnings.

Querel and Rigby [5] made the WarningsGuru source code, a docker installation file, and a running webserver available since 2017, however, there was no substantial adoption. Devanbu *et al.* [27] note that adoption of research tools is difficult because developers are reluctant to change their workflows and have biased perspectives based on their experience. As a retrospective, if we were to create a future WarningsGuru prototype we would integrate it into top ranked continuous integration tools like Jenkins and Travis CI. [2] Since there are already plugins for static bug finder tools, such as FindBugs, we would simply need to create a CommitGuru plugin that uses change measures to flag commits in the CI as potentially bug introducing. Studying and adopting our research ideas would be facilitated by the large number of developers already using these CI tools.

> Despite integrating WarningsGuru in to CommitGuru, we saw little adoption of the tool. This negative result indicates that it may be better to integrate into tools that are widely used in developer CI pipelines such as Jenkins rather than into research tools.

VI. Threats to Validity

The CommitGuru measures and logistic regression are simplistic compared to the current state-of-the-art commit risk and bug prediction measures and models. We acknowledge this limitation, however, this is not a threat to the validity of this study because the WarningsGuru model explained even less deviance than the simple CommitGuru model. Clearly the use of more advanced state-of-the-art statistical bug models which outperform CommitGuru will also outperform WarningsGuru at finding bug introducing changes.

Our results may not generalize beyond the projects under study and the static bug finder tools under study. We did, however, choose a wide range of project domains, from web to analytics to databases. We also examined a large number of commits, over 45k.

There are more advanced static bug finder and static analysis techniques than FindBugs. In this paper we compare and combine change measures with warnings on a large number

[2]Ranking of CI tools, accessed February 2021. https://bitbar.com/blog/top-continuous-integration-tools-for-devops/

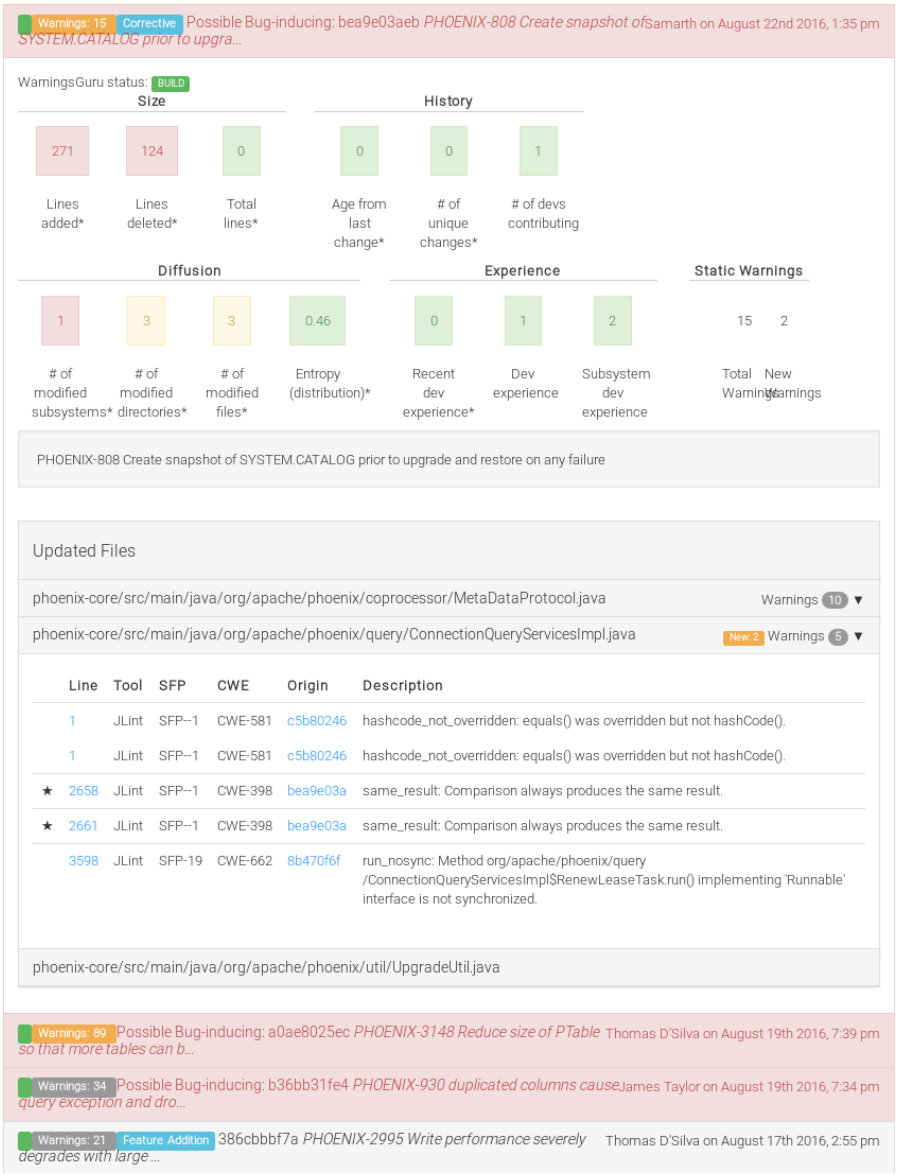

Fig. 4. Integration of WARNINGSGURU in modified COMMITGURU interface. Each commit lists the risk predictors and the warnings associated with each modified file. The individual warnings are shown with links to their originating line on GitHub. A star indicates a new warning.

of commits. As result, we stated our inclusion criteria for both static bug finder tools and projects and select static bug finder tools that could be run in a reasonable time. Future work is necessary to run computationally expensive static analysis tools on large number of projects.

Our primary focus was on assessing the predictive power of WARNINGSGURU warnings in statistical bug models of bug-introducing changes. Instead of coming to the incorrect conclusion, "static bug finders do not find bug introduces changes and are not useful," we conducted a preliminary study of the *perceived* usefulness of warnings (afterall, static bug finder tools are widely used by professional developers [7]). In total we surveyed 179 developers with a response rate of 17.9% and the surveyed developers determined the utility of 81 warnings. While 81 warnings is a relatively small sample, the results were statistically significant due to the large skew in perceived usefulness. Clearly larger studies are necessary, but these results stand as a statistically valid first step.

VII. RELATED WORK

The effectiveness of static bug finders and static analysis in general has long been contentious. A preliminary work by Tang *et al.* [16] on 2 projects and 8 revisions showed that OOP measures such as LCOM "Lack of Coupling in Methods" and McCabe complexity provide less predictive power than static analysis warnings. The predictive models were unstable across revisions suggesting that future work is necessary to replicate these finding on a larger number of project revisions.

Wedyanf *et al.* [28] applied Findbugs, Jlint and the static analysis component of Intellij Idea to 20 releases of 2 projects.

They found that fewer than 3% of warnings are associated with bugs and that the warnings are predominately associated with refactorings.

Couto *et al.* [8] studied the bug finding effectiveness of FindBugs on three projects. Of the 280 bug fixing changes FindBugs produced a warning for only 33. Furthermore, static bug finder tools produce a large number of warnings bewteen 4 and 10 warnings per KLOC depending on the project. Considering the multitude of warnings and the limited effectiveness of identifying commits that contain bugs, the authors find a median precision and recall for FindBugs of zero. The authors conclude that static bug finder warnings do not correspond to field defects. When they examine a individual releases for 30 projects they find a moderate correlations of .56 between the number of warnings and the number of bugs reported against a release.

Work by Rahman *et al.* [3] concluded that "under some accounting principles, they [FindBugs, PMD, JLint] provide comparable benefits [to statistical bug models]." Unfortunately, the methodology and accounting schemes in the paper make replication and interpretation difficult as the statistical models provide predictions at the file level and the static analysis tools provide warnings at the line or code unit level requiring complex "budgeting" of warnings and statistical risk. A further limitation is that the authors process only 34 versions requiring complex git blame assignment of warnings to past revisions ignoring static analysis warnings that may have been removed between releases. Despite these limitations, the authors conclude that the "performance of certain static bug-finders can be enhanced using information provided by statistical defect prediction."

In contrast to these works, our WARNINGSGURU tool runs static bug finder on every commit of a project for a total of 45k commits processed. WARNINGSGURU uses git blame to assign warnings when a commit does *not build*. Since we know exactly when and on which line a warning first was introduced, we eliminate the need for Rahman *et al.*'s "complex accounting." We find that the static analysis measures add only a small 2.5 percentage point increase in deviance explained over the change measures indicating that the computationally expensive static analysis has much redundancy with simple churn measures and will only provide limited enhancement in predictions. Our tool and data are publicly available [4].

Static bug finder tools are widely used in the software industry [7] and proponents of static bug finder correctly argue that the warnings identified are not designed to find many classes of bugs, such as those related to user experience problems. A manual study of FindBugs by at Google [6] found that of the 1127 warnings examined, 17% of the warnings were "impossible" meaning that they "could not be exhibited by the code." An additional 11% of the warnings were deemed to be trivial. While static analysis warnings may not increase the number of bugs found, we have developed a tool to help developers see the warnings that are present in risky commits and those that are new to the current commit. Future user studies are necessary to understand if limiting the number of warnings that a developer sees reduces the effort in eliminating the impossible, trivial, and false positives warnings suggested by static analysis.

VIII. Conclusion and Contributions

We developed tools and techniques to build historical commits and run static bug finders on the build artifacts. In Section II, we used the WARNINGSGURU pipeline to process over 45k commits on 8 projects, and introduced new strategies including identifying the correct library by finding the version that existed closest to the commit data. We were able to double the per project buildability compared to prior work to 78.4% of commits [1]. We also found that 58.5% of commits contain a warning, that 12.8% of all commits introduced new warnings, and 6.8% of all commits have new security warnings. WARNINGSGURU is capable of managing multiple concurrent Maven based projects and identifies the new warnings which are introduced in each commit.

We add the static bug finder warning counts and types to COMMITGURU's statistical bug prediction models and found that the warnings were are largely redundant with simple historical change measures. The static bug finder warnings are ineffective at predicting bug-introducing commits with the model only explaining 13% of the deviance. This negative result contradicts prior work that found under certain accounting schemes that statistical bugs models and static bug finders were similarly effective at identifying bugs [3].

Static bug finders were not designed to find all bugs and are in wide use, so we expanded the analyse to an additional 37 projects for a total of over 55k commits to conduct a preliminary user study of the perceived usefulness of static bug finder warnings. We surveyed 179 developers with a response rate of 17.9% for a total of 81 warnings across 45 days. From these responses we found that the perceived usefulness of the warning is partially dependent on the tool which generated it. 52.9% of new Findbugs warnings were deemed useful as opposed to only 21.3% of new Jlint warnings. The dynamic survey software which includes the commit, warning, and records the developer's response is publicly available [4].

One of the goals of COMMITGURU [10] was to provide "just-in-time" commit risk information to developers. In our user study we evaluated the timeliness of informing the developer of a new warning by evaluating the impact of time on the usefulness of warnings. We concluded that developers who are informed of the warning in a median of 11.5 days are more likely to indicate it as useful as oppose to a median of 23 day for warnings that were not useful. While future work is necessary, this finding quantifies how quickly developers forget the context of a commit.

We integrate the results of WARNINGSGURU into the interface of COMMITGURU [10] where we present the new and historical warnings identified in each commit. These warnings provide concrete actionable insights into why a commit may be flagged as risky. The tool [5] and data [4] are publicly available for both researchers and developers who may be interested in warning-introducing commits.

References

[1] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[2] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," vol. 39, no. 6. Piscataway, NJ, USA: IEEE Press, Jun. 2013, pp. 757–773. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.70

[3] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 424–434. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568269

[4] L.-P. Querel and P. C. Rigby, "WarningsGuru, research scripts and data for replication," https://doi.org/10.5281/zenodo.3747582.

[5] ——, "WarningsGuru tool GitHub Repo," https://github.com/louisq/warningsguru.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251536

[7] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 470–481.

[8] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," vol. 21, no. 2, 2013, pp. 241–257. [Online]. Available: http://dx.doi.org/10.1007/s11219-011-9172-5

[9] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 15, May 2005. [Online]. Available: https://doi.org/10.1145/1082983.1083147

[10] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 966–969. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803183

[11] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," vol. 38, no. 6, Nov 2012, pp. 1276–1304.

[12] L.-P. Querel and P. C. Rigby, "Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 892895. [Online]. Available: https://doi.org/10.1145/3236024.3264599

[13] The Apache Software Foundation, "Maven - POM Reference," 2016, https://maven.apache.org/pom.html.

[14] KDM Analytics, "Blade Tool Output Integration Framework (TOIF)," 2016, http://www.kdmanalytics.com/toif/.

[15] MITRE Corporation, "Common Weakness Enumeration (CWE)," 2016, https://cwe.mitre.org/.

[16] H. Tang, T. Lan, D. Hao, and L. Zhang, "Enhancing defect prediction with static defect analysis," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, ser. Internetware '15. New York, NY, USA: ACM, 2015, pp. 43–51. [Online]. Available: http://doi.acm.org/10.1145/2875913.2875922

[17] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810310

[18] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.23

[19] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," vol. 11, no. 3. New York, NY, USA: ACM, Jul. 2002, pp. 309–346. [Online]. Available: http://doi.acm.org/10.1145/567793.567795

[20] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, Sep. 2014. [Online]. Available: https://doi.org/10.1145/2594458

[21] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985456

[22] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[23] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality?: An empirical case study of windows vista," vol. 52, no. 8. New York, NY, USA: ACM, Aug. 2009, pp. 85–93. [Online]. Available: http://doi.acm.org/10.1145/1536616.1536639

[24] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," vol. 29, no. 6. Piscataway, NJ, USA: IEEE Press, Jun. 2003, pp. 481–494. [Online]. Available: http://dx.doi.org/10.1109/TSE.2003.1205177

[25] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? a study of the chromium project," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 269–279.

[26] L. Averell and A. Heathcote, "The form of the forgetting curve and the fate of memories," vol. 55, no. 1. Elsevier, 2011, pp. 25–35.

[27] P. Devanbu, T. Zimmermann, and C. Bird, "Belief evidence in empirical software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 108–119.

[28] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 141–150. [Online]. Available: http://dx.doi.org/10.1109/ICST.2009.21