Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects

Pavneet Singh Kochhar, David Lo, Julia Lawall, Nachiappan Nagappan

Code Coverage and Post-Release Defects: A Large Scale Study on Open Source Projects

Pavneet Singh Kochhar, David Lo, *Member, IEEE,* Julia Lawall, *Member, IEEE,* and Nachiappan Nagappan

***Abstract*—Testing is a pivotal activity in ensuring the quality of software. Code coverage is a common metric used as a yardstick to measure the efficacy and adequacy of testing. However, does higher coverage actually lead to a decline in post-release bugs? Do files that have higher test coverage actually have fewer bug reports? The direct relationship between code coverage and actual bug reports has not yet been analysed via a comprehensive empirical study on real bugs. Past studies only involve a few software systems or artificially injected bugs (mutants).**

In this empirical study, we examine these questions in the context of open-source software projects based on their actual reported bugs. We analyze 100 large open-source Java projects and measure the code coverage of the test cases that come along with these projects. We collect real bugs logged in the issue tracking system after the release of the software and analyse the correlations between code coverage and these bugs. We also collect other metrics such as cyclomatic complexity and lines of code, which are used to normalize the number of bugs and coverage to correlate with other metrics as well as use these metrics in regression analysis. Our results show that coverage has an insignificant correlation with the number of bugs that are found after the release of the software at the project level, and no such correlation at the file level.

Keywords*—*Empirical study, code coverage, sofware testing, post-release defects, open-source

I. Introduction

Testing is widely believed to be a cornerstone in ensuring software reliability in practice. The increasing size and complexity of software has necessitated improvements in software testing. Nevertheless, testing is expensive, and thus software developers and product managers must constantly address the question of how much testing is enough. A commonly accepted metric is the notion of code coverage. A set of tests is considered adequate when running the tests causes every line, branch, condition, or path, depending on the kind of coverage desired, to be executed at least once. Nevertheless, achieving adequate coverage does not prove that the code is correct. Indeed, every programmer knows that a particular sequence of instructions can produce the expected result on one set of input values and an incorrect result on another. This thus raises the question of whether coverage is actually an accurate predictor of the number of post-release bugs.

Several studies have investigated the correlation between code coverage and test suite effectiveness, measured in terms of number of post-release defects or ability to kill the mutants. Mockus et al. [30] study the correlation between code coverage and post-release bugs on two large industrial projects, Microsoft Windows Vista and a call center reporting system from Avaya. The results of their study did not show a conclusive relationship between coverage and quality. Further, these results cannot be generalized as the projects were developed in a controlled environment and represent only two large, but real-world, applications. Recent studies by Inozemtseva et al. [18] and Gopinath et al. [13] analyse the correlation between coverage and test suite effectiveness. Both these studies use artificially injected bugs, also known as mutants, and measure the effectiveness of a test suite by its ability to kill the mutants. However, empirical research shows that mutants are not representative of real faults [14].

In order to study the relation between coverage and post-release bugs in a broader range of development contexts, we compare coverage rates and the number of post-release bugs in open-source software. Open-source projects are different from closed source projects in terms of decision making, motivation, environment, testing processes and release management [4]. We want to understand if open source projects exhibit similar or different results as compared to those observed in closed source industrial projects. To the best of our knowledge, ours is one of the largest empirical studies that analyzes the impact of coverage on post-release bugs in open-source software.

In this study, we examine 100 large open-source Java projects that use the JIRA[1] bug tracking service, that provides support for bug tracking and project management. We download these 100 projects that are hosted on GitHub and use Maven. GitHub is one of the largest software repositories, which hosts millions of software projects including some popular projects such as spring-roo[2] from Spring, the WildFly Application server[3] (previously JBoss application server) from the WildFly community, and Maven[4] from Apache, all of which are present in our dataset. We execute test cases and calculate coverage for our 100 projects, considering cases where a method is called either directly or indirectly by a test case, and examine the relation between code coverage and the number of bugs found after the release of the software. We then assess the projects in terms of several important software metrics, such as the number of lines of code and the cyclomatic complexity, to understand the effect of these metrics on the correlation between coverage and the number of bugs. We chose these software metrics as they are used to assess the cost of development processes and to evaluate the quality of software [10].

[1] https://www.atlassian.com/software/jira
[2] https://github.com/spring-projects/spring-roo
[3] https://github.com/wildfly/wildfly
[4] https://github.com/apache/maven

We investigate these research questions:

RQ1: What is the correlation between code coverage and the number of post-release bugs at the project level?

RQ2: What is the correlation between code coverage and the number of post-release bugs at the file level?

We make the following contributions:

1) We perform one of the largest studies on open-source Java projects with the aim of studying the impact of code coverage on the number of real bugs found after the release of the software.
2) We measure the test adequacy by executing these test cases and analyse the correlation between code coverage and post-release bugs at the project and file level.
3) We draw on statistical methods and graphs to understand the impact of metrics such as lines of code and cyclomatic complexity on the correlation between code coverage and post-release bugs.
4) We make our dataset publicly available for other researchers to replicate our experiments and conduct future studies.

In this paper, we describe code coverage, and the tools we use to collect information from our dataset in Section II. We explain the methodology of our study in Section III. We perform several statistical tests on the data to answer the two research questions and we provide results for these tests in Section IV. In Section V and Section VI, we provide several threats to validity and related work, respectively. We conclude and mention future work in Section VII.

II. Preliminaries

In this section, we review the definition of code coverage and present the tools that are relevant to our chosen software and our experiments. We use Sonar for collecting software metrics, Sonar relies on Maven for building packages, and we use JIRA for collecting post-release bug information. All of our projects come from GitHub.

A. Code Coverage

Software testing is used to test different functionalities of a program or system and to ensure that given a set of inputs the system produces the expected results. A *test adequacy criterion* defines the properties that must be satisfied for a thorough test [12]. *Code coverage*, which measures the percentage of code executed by test cases, is often used as a proxy for test adequacy. The percentage of code executed by test cases can be measured according to various criteria, including the percentage of executed source code lines (*line coverage*), and the percentage of executed branches (*branch coverage*). Sonar combines these measures into a hybrid measure, referred to as *coverage*.[5] This coverage measure is efficient to compute, while still incorporating information about branches, which are important, because they may lead the program to very different behaviors. We primarily focus on coverage in our experiments.

[5]http://docs.codehaus.org/display/SONAR/Metric+definitions

B. Sonar

Sonar[6] is an open-source platform that helps to the manage software quality of a project. Sonar can either be used as a standalone web based application or can be integrated into a Web Application Container such as Tomcat. Sonar uses various tools, such as JavaNCSS,[7] JaCoCo,[8] Cobertura,[9] and Surefire,[10] to extract software metrics such as cyclomatic complexity, lines of code (LOC), number of test cases, and code coverage.

In our empirical study, we collect software metrics, such as cyclomatic complexity, lines of code, and code coverage using Sonar.

C. Maven

Maven[4] is a software project management tool that supports building and running the software and its test cases. Maven uses information that is present in the project object model (POM) file, *pom.xml*. The POM file contains information about the project such as its dependencies on libraries and the order in which the different components of the project should be built. Maven primarily supports Java projects and for such projects it dynamically downloads all dependencies from a central Maven repository. Sonar makes use of Maven's project directory structure to get various information, such as the number of classes, the number of test cases, the number of packages and the overall lines of code. It also uses this structure to run test cases to collect the coverage of the project.

D. JIRA

JIRA[1] is a project tracker used for issue tracking, bug tracking and efficient project management. To be able to uniformly obtain bug information for the different projects in our dataset, we focus on projects that use JIRA for reporting bugs. For each bug, JIRA records the affected and fixed version of the software, which represent the version in which bug was found and the version in which bug was fixed or resolved, respectively. This information ensures that we are collecting only post-release bugs i.e., those bugs logged after the release of the particular version of the software. We collect information about all the closed and resolved bugs for a particular affected version of the software. JIRA also assigns each bug an identifier that is unique for the given software project. When developers mention this identifier in the logs of the commits that fix the bug, we are able to track the files that were changed to solve the problem.

E. GitHub

GitHub is one of the largest project-hosting platforms and uses the git[11] version control system. GitHub is similar to

[6]http://www.sonarsource.org/

[7]http://www.kclee.de/clemens/java/javancss/

[8]http://www.eclemma.org/jacoco/

[9]http://cobertura.sourceforge.net/

[10]http://maven.apache.org/surefire/maven-surefire-plugin/

[11]http://git-scm.com/

a social network, where software developers spread across the globe can collaborate. Currently, GitHub has more than 11 million users and over 28 million repositories. We clone the repositories of software projects using the command *git clone* {*url*}. We only download projects that contain a Maven *pom.xml* file, implying that they are compatible with Sonar.

III. Methodology & Statistics

In this section, we describe the methodology we use to collect data for this study. Furthermore, we also present several statisics to describe our dataset.

A. Methodology

a) Project Information: First, we search for open-source projects that use JIRA issue tracking system and allow public access to all of the issues filed in the tracking system. We find several examples of projects using public instances of JIRA[12] such as projects developed by the Apache Foundation, Spring Project, the WildFly (formerly JBoss) Community, etc. While these projects are popular and have a large base of contributors, they also cover a wide variety of programs ranging from build management, database, big data, etc. For this, we had to manually find the official web page of each project (>300) and verify whether the project's source code is available on GitHub and to identify their JIRA name. We further restricted the projects to those that use Maven for project management. We, then, collect the source code of projects that are hosted on GitHub and use JIRA issue tracking system. For each project, we visited its website to confirm the major and stable releases and checked out the latest release of the software that was made at least 6 months prior to the month of collection of data (August 2013). For some of the projects, the stable release was made one or two years before August 2013, which gave ample time for users to use the release and report bugs. After collecting the releases, for each one of them, we run Sonar on these projects to collect metrics such as LOC, cyclomatic complexity, code coverage etc. We filter out projects with less than 5,000 lines of code as these projects are small and do not contain many test cases and have even fewer numbers of bugs. In the end, we select top 100 projects sorted by size. Our dataset contains projects of different sizes ranging from 5,000 LOC to more than 100,000 LOC.

Initially, to set up the project, we use the command $mvn\,clean\,install$ in the root of each project repository. The *clean* command removes any files compiled during the prior builds that might be present in the repository and the *install* command builds a dependency tree for all the components specified in the *pom.xml* (the root POM). The *install* command also compiles the *.java* files present in the components specified in *pom.xml* into corresponding *.class* files.

After the install phase, we use the command *mvn sonar:sonar* to collect coverage and other metrics. Before running this command, we need to start the Sonar web server, which has its own Maven repository, data repository, web services and Sonar plug-ins. The Sonar web server synchronizes its Maven repository with the Maven repository of the user where all the artefacts are stored. *mvn sonar:sonar* is used to to make Sonar perform dynamic analysis, i.e., running test cases and creating reports. After the analysis, the reports are published in the repository of the Sonar server, which can be accessed at the default address *http://localhost:9000/*.

Bug collection (Project Level): For each bug, JIRA records the affected version of the software. We collected all of the closed and resolved bugs for the checked out version of the software. We perform this step manually for each software project, as each project has a unique name used by JIRA and each project has a different checked out version. We obtained the JIRA name of each project by searching the project's website. For example, the project Twitter4J[13] in our dataset, for which we use version 3.0.0, has JIRA name *TFJ*.

Bug collection (File Level): For each bug at the project level, we collect the bug key assigned by JIRA, which is unique for given repository. For example, one of the bugs in *Twitter4J* has a key *TFJ-730*. Then, we search the git logs to find all the commits associated with the bug key, and from these commits, we collect the changed files. A single commit can also fix multiple ($n > 1$) bugs. In this case, the number of bugs for the file affected by that commit is n.

b) Statistical Tests: We use commonly accepted statistical analysis to find the correlation between the collected software metrics and the code coverage.

Spearman's rho: Spearman's rank correlation coefficient (ρ) is a non-parametric test that is used to measure the strength of monotonic relationship between sets of data [34]. The value of rho ranges from -1, which signifies a perfect negative correlation, to +1, which signifies a perfect positive correlation. The value 0 shows that there is no correlation between the variables. To calculate Spearman's rho, the raw values from the data sets are arranged in ascending order and each value is assigned a *rank* equal to its position in the list. The values that are identical in two sets are given a rank equal to the average of their positions. Equation 1 then shows the formula for the calculation:

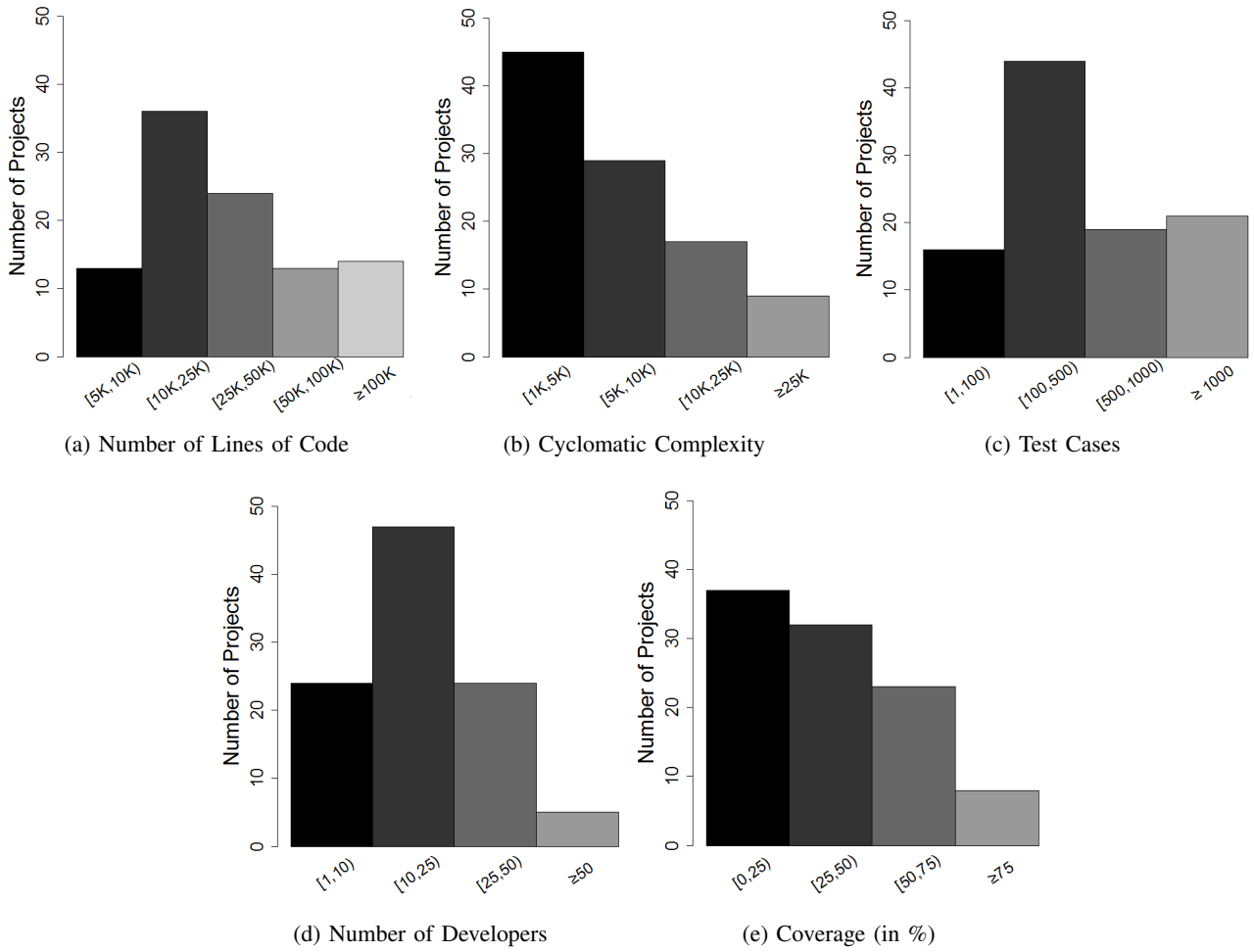
$$\rho = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \quad (1)$$

In this equation, x_i and y_i represent the ranks of input elements X and Y, while $\overline{x}$ and $\overline{y}$ represent the averages of the ranks. We use the following values to interpret correlation [17]: $0 \leq \rho < 0.1$ = None, $0.1 \leq \rho < 0.3$ = Small, $0.3 \leq \rho < 0.5$ = Moderate, $0.5 \leq \rho < 0.7$ = High, $0.7 \leq \rho < 0.9$ = Very High, $0.9 \leq \rho \leq 1.0$ = Perfect.

Kendall's tau: Kendall's rank correlation coefficient (τ) is a non-parametric test for statistical dependence between two sets

[12] https://confluence.atlassian.com/display/JIRAHOST/Examples+of+Public+JIRA+Instances

[13] https://github.com/yusuke/twitter4j



(a) Number of Lines of Code (b) Cyclomatic Complexity (c) Test Cases

(d) Number of Developers (e) Coverage (in %)

Fig. 1: Distribution of Projects

of data [21]. Similar to Spearman's rho, the value of tau ranges from +1 to -1, with 0 signifying no correlation. To calculate Kendall's tau, let (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) be a set of observations for variables X and Y. A pair (x_i,y_i) and (x_j,y_j) is concordant if ranks for both elements agree, i.e., $x_i > x_j$ and $y_i > y_j$ or if $x_i < x_j$ and $y_i < y_j$. The pair is discordant if $x_i > x_j$ and $y_i < y_j$ or if $x_i < x_j$ and $y_i > y_j$. Equation 2 shows the formula for calculating tau:

$$\tau = \frac{(n_c - n_d)}{\sqrt{n_x n_y}} \tag{2}$$

where,
n_c = Number of concordant pairs
n_d = Number of discordant pairs
n_x = Number of pairs with different x values
n_y = Number of pairs with different y values

We use the following ranges to interpret Kendall rank correlation: $0 \leq \tau < 0.1$ = None, $0.1 \leq \tau < 0.3$ = Weak, $0.3 \leq \tau < 0.5$ = Moderate, $0.5 \leq \tau \leq 1.0$ = Strong. Same scale has been used in past software engineering studies [5].

P-value: The p-value is the probability of obtaining a result equal to or more extreme than what was actually observed, when the null hypothesis (H_0) of a study question is true. The significance level (α) refers to a pre-selected value of probability. If p-value is less than the significance level (α), then we can reject the null hypothesis i.e., our sample gives reasonable evidence to support the alternative hypothesis (H_1). In this study, we select the value of α as 5% or 0.05 and if p-value is less than 0.05, we reject the null hypothesis.

All the statistical analysis was performed using R, which is a programming language and software environment for statistical computing that is widely used in academia and industry. To compute Spearman's ρ, we use the equation, *cor.test(x,y, method="spearman")*, where cor.test is provided by the stats package in R, and x and y are numeric vectors of data values of the same length. To compute Kendall's τ, we use the equation *Kendall(x,y)*, where Kendall is provided by

the Kendall package in R, and x and y are numeric vectors of data values of the same length.

B. Statistics

In this section, we present some statistics describing the data we collected for this study. We also provide the values of the project-level statistics characterizing our dataset.

a) Lines of code (LOC): We used Sonar to count the total number of lines of code in each project. Sonar excludes blank lines, comments and test cases while calculating LOC. Figure 1a shows the distribution of the LOC for the projects in our dataset. 13 projects have between 5,000 and 10,000 LOC, 36 projects have between 10,000 and 25,000 LOC, 24 have projects between 25,000 and 50,000 LOC, 13 projects have between 50,000 and 100,000 LOC and 14 projects have more than 100,000 LOC. The largest project in our dataset contains 237,938 LOC.

b) Cyclomatic complexity (CC): Cyclomatic complexity measures the number of linearly independent paths in the source code of a software application [29]. This measure increases by 1 whenever a new method is called or when a new decision point is encountered, such as an if, while, for, &&, case etc. Cyclomatic complexity is often useful in knowing the number of test cases that might be required for independent path testing [38] and a file or project with low complexity is usually easier to comprehend and test [11].

Figure 1b shows the distribution of cyclomatic complexity. Our dataset has 45 projects with complexity between 1,000 and 5,000, 29 projects with complexity between 5,000 and 10,000, 10,000. 17 projects with complexity between 10,000 and 25,000 and 9 projects with complexity above 25,000. The highest value of complexity is 55,940.

c) Test Cases: Sonar also gives information about the total number of test cases in each project, which includes the number of test cases that passed and the test cases that failed. Sonar also provides the number of test cases that were skipped. A test case could be skipped due to missing dependencies, compilation errors, etc.

Figure 1c shows the distribution of test cases in our dataset. The graph shows all the test cases present in the project including the skipped and failing tests. 16 projects have fewer than 100 test cases, 44 projects have between 100 and 500 test cases, 19 projects have between 500 to 1,000 test cases, and 21 projects in our dataset have more than 1000 test cases. The number of test cases in our dataset varies from 1 to 9,390. The mean and the median number of test cases per project are 907.1 and 359.5, respectively.

d) Developer contributions: We use *git log*, which contains the commit history of the project, to get the number of developers who have contributed to the project. Figure 1d shows the distribution of the number of developers. Our dataset has 24 projects with ≥ 1 and < 10 developers and the same number of projects with 25 and 50 developers. 47 projects have 10 or more but less than 25 developers and 5 projects have more than 50 developers. The mean and median numbers of developers are 19.9 and 32, respectively.

e) Coverage: Sonar provides information of the overall coverage for the project. Figure 1e shows the distribution of coverage across all the projects in our dataset. 37 projects have less than 25% coverage, 32 projects have coverage between 25% and 50%, 23 projects have coverage between 50% and 75% and 8 projects have greater than 75% coverage.

f) Efferent couplings (EC): Efferent couplings is a measure of the number of classes used by a specific class. Coupling between classes can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions. A large value of efferent coupling indicates that the stability of one class is dependent on the stability of other classes and makes the software a tightly coupled system, which is difficult to maintain, test and reuse [35].

g) Delta: Delta represents the number of changes made to the files during the development of the particular version of the software. Classes that are changed more often have a higher value of delta and are usually unstable [39]. Delta has been found to be a better predictor of the number of faults than other metrics such as lines of code [16]. We use git tags to find all the tags of a repository and check the website of the project to find the stable version immediately preceding the version that we have selected for our dataset. Then, we collect all of the commits between the previous stable version and the chosen one. Based on these commits, we collect all the files that were changed between these two versions. The number of changes to a file is then the number of times the file is checked in by different commits. Finally, we normalize the number of changes to a file (or the number of commits that touch a file) by the number of months between current version and previous stable version. We do this in order to remove any biasing in a project since each project has a different time gap between the current and previous version.

IV. Findings

In this section, we investigate our research questions and present the results.

A. RQ1: Coverage & Defects (Project Level)

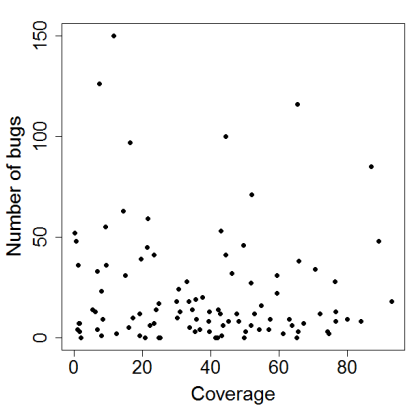
In this question, we investigate the correlation between code coverage and post-release defects at the project level.

Motivation: Code coverage gives us an idea of the thoroughness of testing by providing information about the amount of code that is tested. Increasing coverage, however, requires more work in terms of test case development, and may also increase the test suite running time. Thus, it is useful to understand whether an increase in code coverage is likely to lead to a decrease in post-release bugs.

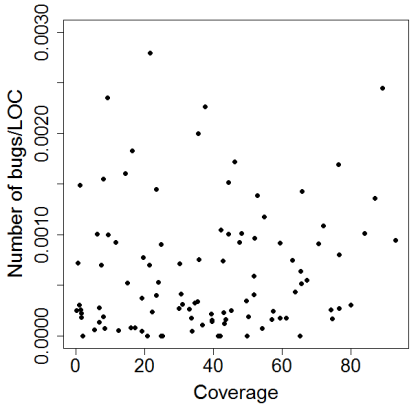
Methodology: We calculate lines of code, coverage, cyclomatic complexity and efferent couplings values by running Sonar for every release. We analyze the projects' JIRA bug repositories to calculate the number of post-release bugs. The detail on how the number of post-release bugs is computed

TABLE I: Distribution of bugs, test cases and coverage.

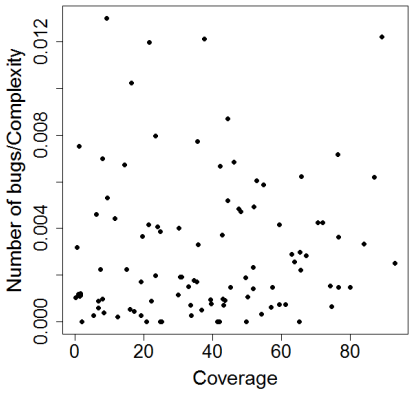
Lines of Code (LOC)	Number of Projects	Number of Bugs (Average)	Number of Test Cases (Average)	Code Coverage (Average)
$\geq$5,000 – <10,000	13	5.769	236.000	40.654
$\geq$10,000 – <25,000	36	14.250	484.361	44.389
$\geq$25,000 – <50,000	24	16.958	450.500	35.425
$\geq$50,000 – <100,000	13	44.615	957.077	32.792
$\geq$100,000	14	49.357	3354.214	26.714



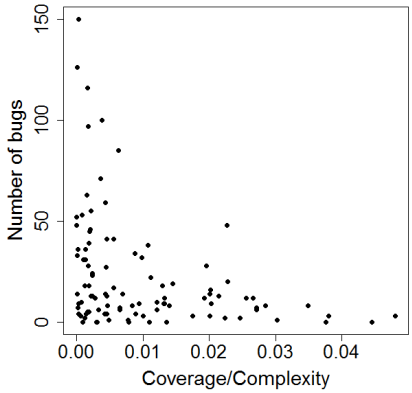
(a) Coverage vs. Number of bugs
ρ=-0.059, p-value=0.559
τ=-0.043, p-value=0.531



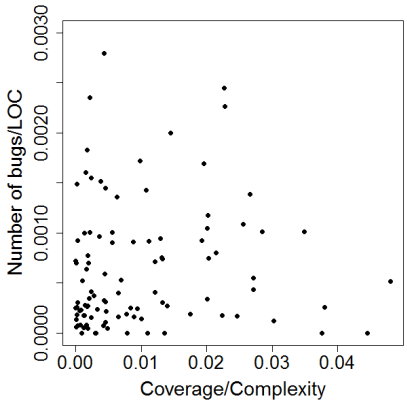
(b) Coverage vs. Number of bugs/LOC
ρ=0.157, p-value=0.117
τ=0.105, p-value=0.124



(c) Coverage vs. Number of bugs/Complexity
ρ=0.139, p-value=0.168
τ=0.086, p-value=0.203



(d) Coverage/Complexity vs. Number of bugs
ρ=-0.359, p-value=0.0002
τ=-0.253, p-value=0.0002



(e) Coverage/Complexity vs. Number of bugs/LOC
ρ=0.175, p-value=0.082
τ=0.116, p-value=0.089

Fig. 2: Spearman's (ρ) and Kendall's (τ) correlations with p-values at the project level

by analyzing JIRA repositories is provided in Section III-A. We derive additional metrics such as number of bugs/LOC, number of bugs/complexity and coverage/complexity. We then compute correlations between coverage and various metrics to answer this research question.

Findings: First, we report the total number of bugs present in the projects segregated based on the lines of code (Table

TABLE II: Spearman's (ρ) and Kendall's (τ) correlations between coverage and different metrics at the project level for 3 categories: small size, medium size and large size projects.

	Correlations	ρ	p-value	τ	p-value
Small Size Projects (<13,562 LOC)	Coverage vs. Number of bugs	0.084	0.691	0.038	0.814
	Coverage vs. Number of bugs/LOC	0.170	0.418	0.101	0.497
	Coverage vs. Number of bugs/Complexity	0.124	0.554	0.061	0.691
	Coverage/Complexity vs. Number of bugs	-0.143	0.496	-0.127	0.397
	Coverage/Complexity vs. Number of bugs/LOC	-0.009	0.965	-0.034	0.833
Medium Size Projects (≥13,562 & <52,890 LOC)	Coverage vs. Number of bugs	0.005	0.973	0.007	0.953
	Coverage vs. Number of bugs/LOC	0.049	0.733	0.040	0.688
	Coverage vs. Number of bugs/Complexity	0.024	0.870	0.017	0.867
	Coverage/Complexity vs. Number of bugs	-0.039	0.790	-0.030	0.769
	Coverage/Complexity vs. Number of bugs/LOC	0.115	0.425	0.079	0.422
Large Size Projects (≥52,890 LOC)	Coverage vs. Number of bugs	0.135	0.521	0.097	0.513
	Coverage vs. Number of bugs/LOC	0.205	0.323	0.127	0.388
	Coverage vs. Number of bugs/Complexity	0.243	0.241	0.160	0.272
	Coverage/Complexity vs. Number of bugs	-0.020	0.926	0.017	0.926
	Coverage/Complexity vs. Number of bugs/LOC	0.348	0.088	0.267	0.065

TABLE III: Spearman's (ρ) and Kendall's (τ) correlations between coverage and different metrics at the project level for low and high complexity projects.

	Correlations	ρ	p-value	τ	p-value
Low Complexity Projects (<5,713)	Coverage vs. Number of bugs	0.005	0.974	-0.001	1.000
	Coverage vs. Number of bugs/LOC	0.074	0.611	0.053	0.598
	Coverage vs. Number of bugs/Complexity	0.030	0.835	0.007	0.953
	Coverage/Complexity vs. Number of bugs	-0.231	0.107	-0.175	0.080
	Coverage/Complexity vs. Number of bugs/LOC	-0.059	0.682	-0.043	0.663
High Complexity Projects (≥5,713)	Coverage vs. Number of bugs	-0.025	0.865	-0.014	0.893
	Coverage vs. Number of bugs/LOC	0.137	0.341	0.085	0.389
	Coverage vs. Number of bugs/Complexity	0.136	0.348	0.092	0.353
	Coverage/Complexity vs. Number of bugs	-0.274	0.054	-0.185	0.061
	Coverage/Complexity vs. Number of bugs/LOC	0.123	0.394	0.080	0.417

I). We can observe that the number of bugs increases with the size of the projects. The 13 projects having size between 5,000 to 10,000 LOC have 75 reported bugs, whereas the 13 projects present in the range 50,000 to 100,000 LOC have 580 reported bugs. The 14 projects having size above 100,000 LOC have the largest number of reported bugs, 691.

Next, we analyse the correlation between the amount of code coverage and the number of bugs. We want to determine whether the number of post-release bugs decreases with an increase in the coverage of the software. Our null hypothesis is that there is no significant correlation between coverage and number of bugs, whereas the alternate hypothesis is that there is a significant correlation between these two variables. Figure 2a depicts the correlation between code coverage and the number of bugs. The coverage levels for all the projects span from 0.1% to 93% with an average value of 37.76%. From the figure, we can observe that as the coverage increases, there is no reduction in the number of bugs. The Spearman's ρ value is -0.059 (p-value=0.559) and Kendall's τ value is -0.043 (p-value=0.531), which shows that there is a statistically insignificant correlation (p-value>0.05) between code coverage and the number of bugs. As such, we cannot reject the null hypothesis.

Since our data set consists of projects that are of varying size and complexity, we divide the number of bugs by the number of LOC and complexity to more fairly compare the different projects. We perform a correlation to analyse the impact of coverage on the number of bugs normalized by metrics (LOC and complexity). The null hypotheses are that there are no significant correlations of coverage with number of bugs/LOC and number of bugs/complexity, while the alternate hypotheses state that there are significant correlations between coverage and these metrics. Figures 2b and 2c show the correlation between coverage and these metrics. The Spearman's ρ and Kendall's τ for coverage vs. number of bugs/LOC (ρ=0.157, p-value=0.117; τ=0.105, p-value=0.124) and number of bugs/complexity (ρ=0.138, p-value=0.168; τ=0.086, p-value=0.203) show insignificant correlations between the number of bugs/LOC and the number of bugs/complexity with code coverage. Thus, we cannot reject the null hypotheses.

Further, we define a new metric called normalized coverage where we divide the coverage level of a project by its cyclomatic complexity. This allows more fairly comparing

projects having the same coverage but different complexity values. Our previous study [26] shows that larger as well as more complex projects exhibit low coverage, whereas smaller and less complex projects have higher coverage. As projects with higher complexity are commonly considered to be more difficult to test, if two projects have the same coverage level, their relative complexity reflects the amount of effort put in by developers during testing to achieve that coverage value. We define null hypotheses in this case as: there are no significant correlations between coverage/complexity with number of bugs and coverage/complexity with number of bugs/LOC. The alternate hypotheses are that there are significant correlations between coverage/complexity with number of bugs and coverage/complexity with number of bugs/LOC. Figure 2d and 2e show the correlation of normalized coverage with the number of bugs and the number of bugs/LOC, respectively. The graph shows that the number of bugs decreases with the increase in the value of normalized coverage. The Spearman's ρ and Kendall's τ values are -0.359 (p-value=0.0002) and -0.253 (p-value=0.0002), respectively, which shows a moderate negative correlation between normalized coverage and the number of bugs. However, there is an insignificant correlation between normalized coverage and number of bugs/LOC (ρ=0.175, p-value=0.081; τ=0.116, p-value=0.089). Thus, we can reject the null hypothesis for coverage/complexity and number of bugs, but cannot reject the null hypothesis for coverage /complexity and number of bugs/LOC.

To understand the correlations between coverage and various metrics for projects of different sizes, we divide the dataset into different categories based on the project size. We compute quartiles to divide the projects into three categories: those whose size is less than the lower quartile (25% of the projects), those whose size is between the lower and upper quartile (50% of the projects), and those whose size is above the upper quartile (25% of the projects). We name these three categories as: small ($<$13,562 LOC), medium ($\geq$13,562 LOC & $<$52,890 LOC) and large ($\geq$52,890 LOC), respectively. We then compute correlations for each category separately. The null hypotheses are that there are no significant correlations between coverage and various metrics for projects of different sizes, while the alternate hypotheses state that there are significant correlations between coverage and various metrics. Table II shows the Spearman's and Kendall's correlations between coverage and different metrics for the three categories. We observe that the correlations are insignificant (p-value$>$0.05) for all the categories. Thus, we cannot reject the null hypothesis for all the correlations.

To understand the correlations between coverage and various metrics for projects of different cyclomatic complexities, we divide our dataset into two categories based on the median value of cyclomatic complexity: low complexity ($<$5,713) and high complexity ($\geq$5,713). We then compute correlations between coverage and different metrics for each of the two categories. The null hypotheses state that there are no significant correlations between coverage and various metrics for low and high complexity projects. Our alternative hypotheses are that there are significant correlations between coverage and various metrics for projects with low and high complexity. Table III shows the different correlations. From the results, we observe that all the correlations are insignificant (p-value$>$0.05) for all the categories. As such, we cannot reject the null hypotheses.

At the project level, code coverage has an insignificant correlation with the number of bugs as well as with the number of bugs per LOC and the number of bugs per complexity. Coverage/complexity has a moderate negative correlation with the number of bugs and an insignificant correlation with the number of bugs/LOC. By categorizing projects based on size and complexity, we observe an insignificant correlation between coverage and other metrics.

B. RQ2: Coverage & Defects (File Level)

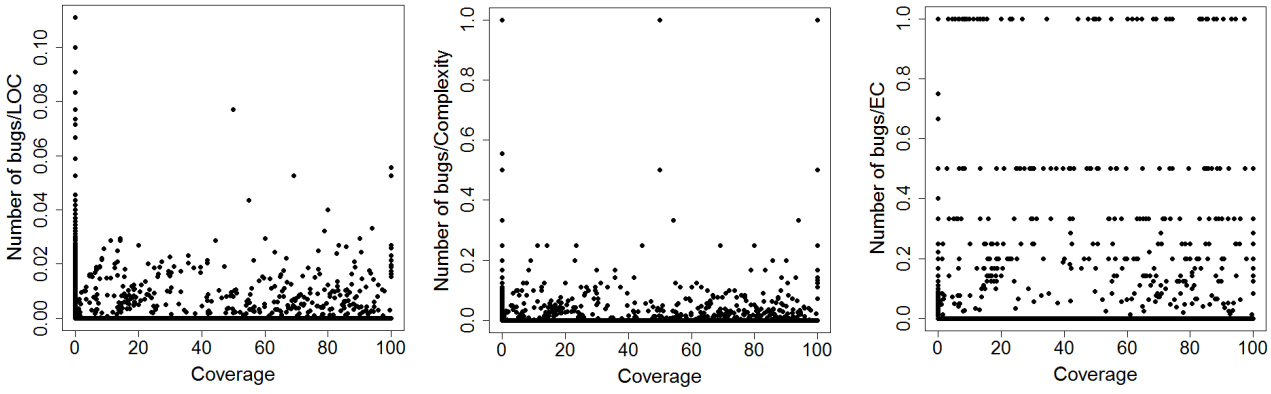
Here, we investigate the correlation between the coverage level of each file and the number of bugs associated with that file. We also assess the number of bugs in terms of other metrics such as cyclomatic complexity, lines of code (LOC) and efferent couplings.

Motivation: The coverage level provides information about the testedness of a project. However, a project may consist of many source code files with diverse properties. Thus, we want to analyse the correlation between coverage and post-release bugs at the file level. Analysing this correlation can enhance our understanding of the impact of coverage on the bugs reported after the release of the software and exhibit which files are adequately tested.

Methodology: We calculate lines of code, coverage, cyclomatic complexity and efferent couplings values by running Sonar for every release. Sonar provides these values for all the files within a release. We analyze the projects' JIRA bug repositories to calculate the number of post-release bugs for each file. The detail on how the number of post-release bugs per file is computed by analyzing JIRA repositories is provided in Section III-A. Similar to the project level, we derive additional metrics such as the number of bugs/LOC, number of bugs/complexity and coverage/complexity. We then compute correlations between coverage and various metrics to answer this research question.

Findings: We normalize the number of bugs by three metrics: lines of code, cyclomatic complexity and efferent couplings. Figure 3a, 3b and 3c show the correlation between coverage and the normalized metrics number of bugs/LOC, number of bugs/Complexity and number of bugs/EC. All three graphs are fitted to the same scale for comparison. We can observe that all the graphs show a similar trend, i.e., there is no correlation between coverage and the other metrics. With the increase in the coverage value, we do not observe a reduction in the number of bugs.

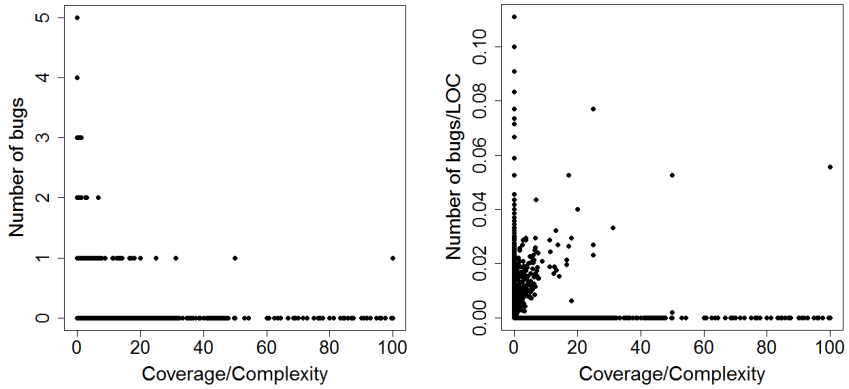
To confirm the behaviour observed in Figure 3a, 3b, 3c, we use Spearman's and Kendall's correlations between coverage and number of bugs/LOC, number of bugs/CC and number of bugs/EC. Our null hypotheses are that there are no significant correlations between coverage and number of bugs/LOC, number of bugs/CC and number of bugs/EC, whereas the alternative hypotheses state that there are significant correlations between coverage and number of bugs/LOC, number



(a) Coverage vs. Number of bugs/LOC
ρ=-0.023, p-value=$1.710e^{-05}$
τ=-0.021, p-value=$1.648e^{-05}$

(b) Coverage vs. Number of bugs/Complexity
ρ=-0.023, p-value=$1.691e^{-05}$
τ=-0.021, p-value=$1.626e^{-05}$

(c) Coverage vs. Number of bugs/EC
ρ=-0.023, p-value=$1.783e^{-05}$
τ=-0.021, p-value=$1.761e^{-05}$



(d) Coverage/Complexity vs. Number of bugs
ρ=-0.030, p-value=$4.034e^{-08}$
τ=-0.026, p-value=$4.026e^{-08}$

(e) Coverage/Complexity vs. Number of bugs/LOC
ρ=-0.029, p-value=$4.453e^{-08}$
τ=-0.026, p-value=$4.904e^{-08}$

Fig. 3: Spearman's (ρ) and Kendall's (τ) correlations with p-values at the file level

of bugs/CC and number of bugs/EC. Table IV shows the correlations among these variables. We can observe that there is no correlation between coverage and any of the other three metrics, however, all the correlations are significant. Thus, we can reject the null hypothesis for all the correlations. This confirms that coverage has no impact on the number of post-release bugs at the file level.

Table V shows the distribution of files segregated based on the different coverage levels and several metrics such as cyclomatic complexity, lines of code and efferent couplings added over all the files. The values in parentheses specify the average values of the respective metrics. The total number of bugs/file for the files having coverage level 0% to 30% is 2.23 times more than the number of bugs/file present in files having coverage over 80%, since the number of files in the range 0% to 30% is very high (2.8 times files with coverage over 80%).

TABLE IV: Spearman's and Kendall's correlations between coverage and software metrics at the file level.

		Number of bugs/LOC	Number of bugs/CC	Number of bugs/EC
Spearman	ρ	-0.023	-0.023	-0.023
	p-value	$1.710e^{-05}$	$1.691e^{-05}$	$1.783e^{-05}$
Kendall	τ	-0.021	-0.021	-0.021
	p-value	$1.648e^{-05}$	$1.626e^{-05}$	$1.761e^{-05}$

The largest number of bugs per file (mean value), largest value of complexity per file and largest value of lines of code per file are in the coverage level 30% to 60%, i.e., 0.031, 33.38 and 140.48, respectively. The maximum value of efferent couplings per file is 5.04 (60% to 80%). We can observe that with the increase in coverage above 30%, the average values of lines of

TABLE V: Counts added over all the class files

Coverage	≥0%,<30%	≥30%,<60%	≥60%,<80%	≥80%
Number of Bugs	588(0.029)	84(0.031)	78(0.021)	91(0.013)
Lines of Code(LOC)	2,186,998(108.20)	384,073(140.48)	502,981(138.03)	633,969(87.95)
Cyclomatic Complexity(CC)	487,234(24.11)	91,270(33.38)	118,305(32.47)	139,952(19.42)
Efferent Couplings(EC)	83,101(4.11)	13,066(4.78)	18,361(5.04)	32,972(4.57)
Files	20,212	2,734	3,644	7,208

TABLE VI: Spearman's correlations among the variables

	Number of Bugs	Lines of Code	Delta	Efferent Couplings	Cyclomatic Complexity	Line Coverage	Branch Coverage
Number of Bugs	1	0.105*	0.141*	0.061*	0.098*	-0.023*	-0.003
Lines of Code		1	0.262*	0.457*	0.927*	-0.013*	0.279*
Delta			1	0.172*	0.260*	0.033*	0.106*
Efferent Couplings				1	0.433*	0.079*	0.184*
Cyclomatic Complexity					1	0.002	0.318*
Line Coverage						1	0.713*
Branch Coverage							1

*$p < 0.05$

TABLE VII: Kendall's correlations among the variables

	Number of Bugs	Lines of Code	Delta	Efferent Couplings	Cyclomatic Complexity	Line Coverage	Branch Coverage
Number of Bugs	1	0.086*	0.132*	0.053*	0.081*	-0.020*	-0.003
Lines of Code		1	0.205*	0.339*	0.795*	-0.010*	0.209*
Delta			1	0.142*	0.206*	0.027*	0.091*
Efferent Couplings				1	0.325*	0.061*	0.149*
Cyclomatic Complexity					1	0.002	0.241*
Line Coverage						1	0.656*
Branch Coverage							1

*$p < 0.05$

code, complexity and couplings decrease. On the other hand, files having 0% to 30% coverage have lower values of lines of code per file, complexity per file and couplings per file than the corresponding values in other coverage levels (30% to 60% and 60% to 80%). This could be due to very large number files having 0% to 30% coverage, i.e., 20,212 which is much higher than the number of files present in other coverage levels.

Table VI and VII show the Spearman's and Kendall's correlations among the variables collected for all the files. The null hypotheses are that there are no significant correlations between various metrics such as Lines of Code and Line Coverage etc. Our alternative hypotheses in this case are that there are significant correlations between various metrics. We can observe that (1) the ρ value for line coverage vs. number of bugs is -0.023 (p-value = $2.732e^{-05}$), (2) the ρ value for branch coverage vs. number of bugs is -0.003 (p-value = 0.590). Similar values are observed for Kendall's correlation. This shows that the number of bugs has no correlation with line coverage and an insignificant correlation with branch coverage. The number of bugs has a small correlation with delta (number of file changes), i.e., 0.121 (p-value $< 2.2e^{-16}$), whereas the number of bugs has no correlation with cyclomatic complexity and efferent couplings. We can reject the null hypothesis for all the correlations except cyclomatic complexity vs. line coverage and number of bugs vs. branch coverage. Our results are contrary to what was observed by Mockus et al. [30]. They found that coverage has a small negative correlation with the post-release defects for the Avaya project and a positive correlation with the post-release defects for Microsoft project. Furthermore, their results show that the number of failures has a strong correlation with lines of code, delta, efferent couplings (which they called FanOut [30]) and cyclomatic complexity, whereas our results show no such correlation between these metrics, except between the number of bugs and efferent couplings, where the correlation is also very small.

A project contains files with different values of complexity and coverage. If we combine complexity and coverage, there can be four different cases: high complexity and low coverage, low complexity and low coverage, high complexity and high coverage and low complexity and high coverage. In the first case, the high complexity suggests that it is difficult to test the file and thus the low coverage means this file should have more bugs. Secondly, when the coverage is low, the file should have a lower number of bugs as compared to first case since the

TABLE VIII: Negative Binomial Regression Model
AIC=7567.55, BIC=7618.11, Log Likelihood=-3777.77, Deviance=4313.76, Number of Observations=33798

	Estimate	Std. Error	z-value	Pr($>\|z\|$)	
(Intercept)	-3.983	0.048	-82.991	$< 2e^{-16}$	***
Cyclomatic Complexity	0.003	0.000	6.340	$2.29e^{-10}$	***
Delta	0.072	0.004	16.050	$< 2e^{-16}$	***
Efferent Couplings	0.017	0.004	3.828	0.000	***
Branch Coverage	-0.003	0.001	-2.739	0.006	**

$^{***}p < 0.001, ^{**}p < 0.01$

complexity is low. Although the complexity is high in the third case, the files having these characteristics should contain fewer bugs than the files in the first two cases, since the coverage is high. In the last case, complexity is low and higher coverage means that these files should have the fewest bugs.

Similar to the project level, we normalize the coverage values of the files with their respective complexity values. Our null hypotheses are that there are no significant correlations between coverage/complexity with number of bugs and coverage/complexity with number of bugs/LOC. The alternate hypotheses are that there are significant correlations between coverage/complexity with number of bugs and coverage/complexity with number of bugs/LOC. Figure 3d shows the correlation between coverage/complexity and the number of post-release bugs found in the class files. We can observe that there is no correlation even after we normalize the coverage by complexity. The Spearman's ρ is -0.030 (p-value = $4.034e^{-08}$) and Kendall's τ is -0.026 (p-value = $4.026e^{-08}$) confirming that there is no correlation between these two metrics. Further, we normalize the number of bugs by lines of code, to make it easier to compare files of different sizes. Figure 3e shows the correlation between the number of bugs per lines of code and normalized coverage. The Spearman's ρ value is -0.029 (p-value= $4.453e^{-08}$) and Kendall's τ value is -0.026 (p-value = $4.904e^{-08}$), which shows that there is no correlation. The correlations are significant, thus, we can reject the null hypotheses for both the cases.

Further, to understand the impact of factors such as coverage, cyclomatic complexity, delta and efferent couplings on the number of post-release bugs, we use a negative binomial regresssion (NBR) model, which is a type of generalized linear model for modeling count variables. NBR is appropriate for our study as it can handle over-dispersion e.g., cases where the variance of the response variable is greater than the mean [9]. We learn a regression model with similar predictor variables as those used by Mockus et al. [30], i.e., delta, efferent couplings and branch coverage. The regression equation is shown in Equation 3. In the equation, β_1, β_2, β_3 and β_4 are the regression coefficients of the predictor variables. They represent the difference in the logs of expected number of bugs for one-unit difference in any one of the predictor variables when all others are held constant. The intercept value (α) shows the expected number of bugs if the predictor variables (i.e., cyclomatic complexity, delta, efferent couplings, and branch coverage) are all zero. However, for our case, the predictor variables are never all zeroes, and thus the intercept value has no intrinsic meaning. It does not tell us any relationship between the predictor variables and the number of bugs. We learn the coefficients of the model by using R; in particular we use glm.nb function provided by the MASS[14] package.

To check for excessive multi-collinearity, we compute the variance inflation factor (VIF) of each dependent variable in our model. We compare the VIF value computed from our data with the commonly used value of VIF equal to 5 [9]. We find that including LOC and complexity in the model leads to a very high value of VIF. Thus, we remove LOC from the model. Similarly, line and branch coverage are strongly correlated to each other, and therefore, we only include branch coverage. Thus, in all, we use the four predictor variables: branch coverage, complexity, efferent couplings and delta to estimate the value of the response variable i.e., the number of post-release bugs. We also performed a Vuong test to compare NBR with other models such as Poisson and find that NBR has a significant improvement over Poisson (p-value=0.000). Thus, we use the NBR model to analyze our data.

$$\begin{aligned}\text{Number of post-release bugs} = \alpha &+ \beta_1 \text{ Cyclomatic Complexity} \\ &+ \beta_2 \text{ Delta} \\ &+ \beta_3 \text{ Efferent Couplings} \\ &+ \beta_4 \text{ Branch Coverage} + \epsilon\end{aligned} \tag{3}$$

Table VIII shows the result of the NBR model. The null hypothesis for regression is that coverage has no significant effect on the number of post-release bugs when all other variables are held constant, whereas the alternative hypothesis is that coverage has an effect on the number of post-release bugs. The values under the Estimate column show the impact of all four factors on the number of post-release bugs. The intercept value (also called as constant) is the expected mean value of response variable, i.e., number of post-release bugs when all the predictor variables are zero. We can read the coefficients as that for one unit change in the predictor variable, with all other predictor variables held constant, the difference in the logs of expected counts of the response variable is expected to change by the value given by the regression coefficient. For example, one unit increase in the value of branch coverage is expected to reduce the logs of the expected count values by 0.003. Thus, one unit increase in branch

[14]https://cran.r-project.org/web/packages/MASS/MASS.pdf

TABLE IX: Spearman's (ρ) and Kendall's (τ) correlations between coverage and different metrics at the file level for 3 categories: small size, medium size and large size projects.

	Correlations	ρ	p-value	τ	p-value
Files in Small Size Projects (<13,562 LOC)	Coverage vs. Number of bugs	0.004	0.843	0.004	0.843
	Coverage vs. Number of bugs/LOC	0.004	0.843	0.004	0.841
	Coverage vs. Number of bugs/Complexity	0.004	0.848	0.004	0.847
	Coverage/Complexity vs. Number of bugs	-0.026	0.237	-0.023	0.237
	Coverage/Complexity vs. Number of bugs/LOC	-0.026	0.239	-0.022	0.240
Files in Medium Size Projects (≥13,562 & <52,890 LOC)	Coverage vs. Number of bugs	-0.053	$2.435e^{-08}$	-0.047	$2.494e^{-08}$
	Coverage vs. Number of bugs/LOC	-0.053	$1.808e^{-08}$	-0.047	$1.770e^{-08}$
	Coverage vs. Number of bugs/Complexity	-0.053	$1.630e^{-08}$	-0.047	$1.578e^{-08}$
	Coverage/Complexity vs. Number of bugs	-0.067	$1.612e^{-12}$	-0.059	$1.720e^{-12}$
	Coverage/Complexity vs. Number of bugs/LOC	-0.067	$1.477e^{-12}$	-0.059	$1.459e^{-12}$
Files in Large Size Projects (≥52,890 LOC)	Coverage vs. Number of bugs	-0.004	0.546	-0.004	0.546
	Coverage vs. Number of bugs/LOC	-0.004	0.545	-0.004	0.545
	Coverage vs. Number of bugs/Complexity	-0.004	0.553	-0.004	0.554
	Coverage/Complexity vs. Number of bugs	-0.006	0.409	-0.005	0.408
	Coverage/Complexity vs. Number of bugs/LOC	-0.006	0.427	-0.005	0.444

TABLE X: Spearman's (ρ) and Kendall's (τ) correlations between coverage and different metrics at the file level for low and high complexity projects.

	Correlations	ρ	p-value	τ	p-value
Files in Low Complexity Projects (<5,713)	Coverage vs. Number of bugs	-0.093	$1.495e^{-13}$	-0.081	$1.696e^{-13}$
	Coverage vs. Number of bugs/LOC	-0.093	$1.001e^{-13}$	-0.081	$1.001e^{-13}$
	Coverage vs. Number of bugs/Complexity	-0.094	$7.751e^{-14}$	-0.082	$7.342e^{-14}$
	Coverage/Complexity vs. Number of bugs	-0.113	$<2.2e^{-16}$	-0.098	$<2.2e^{-16}$
	Coverage/Complexity vs. Number of bugs/LOC	-0.113	$<2.2e^{-16}$	-0.098	$<2.2e^{-16}$
Files in High Complexity Projects (≥5,713)	Coverage vs. Number of bugs	-0.007	0.245	-0.006	0.245
	Coverage vs. Number of bugs/LOC	-0.007	0.240	-0.006	0.239
	Coverage vs. Number of bugs/Complexity	-0.007	0.244	-0.006	0.243
	Coverage/Complexity vs. Number of bugs	-0.010	0.099	-0.009	0.098
	Coverage/Complexity vs. Number of bugs/LOC	-0.010	0.103	-0.009	0.107

coverage will lead to a decrease in the number of bugs by $e^{0.003}$=1.003 or 0.3% change. Our regression results are similar to the findings of Mockus et al. [30], who find that higher coverage is associated with lower number of bugs, however, the effect is very small. Our results show a small yet significant effect of coverage on number of post-release bugs. Thus, we can reject the null hypothesis.

To understand the correlations between coverage and various metrics for files, we divide the dataset into different categories based on the size of the project they belong to. We club files based on the corresponding project size: small (<13,562 LOC), medium (≥13,562 LOC & <52,890 LOC) and large (≥52,890 LOC). We then compute correlations for each category separately. Table IX shows the correlations between coverage and different metrics for the three categories. The null hypotheses in this case are that there are no significant correlations between coverage and various metrics for files present in projects of different sizes, while the alternate hypotheses state that there are significant correlations between coverage and various metrics. We observe that for files present in projects of small and large sizes, the correlations between coverage and different metrics are insignificant. For files in medium projects, we observe no correlation between coverage and different metrics. From the p-values, we can reject the null hypothesis for files in medium size projects, however, we cannot reject the null hypotheses for files in small and large size projects.

To understand the correlations between coverage and various metrics for files of projects with different cyclomatic complexities, we group files based on project complexity. We divide our dataset into two categories based on the median value of project cyclomatic complexity: low complexity (<5,713) and high complexity (≥5,713). We then compute correlations between coverage and different metrics for each of the two categories. The null hypotheses in this case are that there are no significant correlations between coverage and various metrics for files present in low and high complexity projects, while the alternate hypotheses state that there are significant correlations between coverage and various metrics these two categories. Table X shows that there is a small correlation between coverage/complexity and number of bugs, and coverage/complexity and number of bugs/LOC for files present in projects with low complexity. For all other metrics, we observe no correlation between coverage and each metric. On the other hand, for files present in projects with high complexity, we observe that correlation between coverage and each metric is insignificant. Thus, we can reject the null hypothesis for files

in low complexity projects, however, we cannot reject the null hypotheses for files in high complexity projects.

At the file level, coverage has no correlation with the number of post-release bugs, number of bugs/LOC, number of bugs/complexity and number of bugs/efferent couplings. Furthemore, coverage/complexity has no correlation with the number of bugs as well as number of bugs/LOC. From the regression model, we find that the number of bugs decreases with the increase in the value of coverage, although the impact is very small. By categorizing files based on size of the project they belong to, we observe no correlation between coverage and other metrics for files in medium sized projects and insignificant correlation for files in small and large projects. For files present in low and high complexity projects, we observe no and insignificant correlation between coverage and various metrics, respectively.

V. Threats to Validity

In this section, we describe several threats to validity for our empirical study.

External Validity: These threats relate to the generalizability of the results. In this study, we have investigated 100 large and popular open-source Java projects from GitHub. GitHub is a one of the largest repositories and hosts millions of projects of different sizes and from various domains. We have tried to ensure that our dataset consists of projects of substantial size (>5K LOC).

Internal Validity: These threats are related to the environment under which experiments were carried out. We use Sonar to calculate several metrics such as lines of code, cyclomatic complexity, number of test cases and code coverage. Sonar uses Maven's directory structure to calculate these metrics. In this study, we do not consider projects that do not use Maven i.e., they do not contain a *pom.xml* file. It is possible that projects that do not entirely follow Maven's structure may be interpreted wrongly. This could lead to Maven wrongly calculating certain metrics such lines of code or miss test cases in the project, which can affect the coverage value. We have manually checked a few projects and they fully conform to the Maven directory structure. While counting the delta (number of times a file is changed), we use a major version previous to the current checked out version because it is difficult to find the exact previous version in the repository. So, we may have wrongly identified the number of times the files have changed. Furthermore, while collecting bugs at the file level, we used bug keys, which were collected at the project level from JIRA. Some of these bug keys were not mentioned in any of the *git logs*, so we could not identify the files that were changed in order to solve those bugs. That may have led to non-identification of files which were buggy. However, we believe this is a common problem when working with open-source systems since developers are not forced to tag bug fixes according to the bug key.

VI. Related Work

In this section, we describe several past studies on software testing, code coverage and analysis of open-source projects. Our survey is by no means complete.

A. Studies on Testing & Code Coverage

Past studies have analyzed the importance of testing on the overall quality of the software. Our work is closely related to the study conducted by Mockus et al. [30], where they investigate two industrial software projects from Microsoft and Avaya with the goal of understanding the impact of coverage on test effectiveness. They also calculate the amount of test effort required to achieve different coverage levels. Their results show that increasing test coverage reduces field problems but increases the amount of effort required for testing.

Ahmed et al. analyse a large number of systems from GitHub and Apache and propose a novel evaluation of two commonly used measures of test suite quality: statement coverage and mutation score, i.e., the percentage of mutants killed [1]. They compute test suite quality by correlating testedness of a program element (class, method, statement or block) with the number of bug-fixes. They define testedness as how well a program element is tested, which can be measured using metrics such as coverage and mutation score. They find that statement coverage and mutation score have a weak negative correlation with bug-fixes. However, program elements covered by at least one test case have half as many bug-fixes compared to elements not covered by any test case. Cai and Lyu use coverage and mutation testing to analyse the relationship between code coverage and fault detection capability of test cases [7]. Cai performs an empirical investigation to study the fault detection capability of code coverage and finds that code coverage is a moderate indicator of fault detection when used for all the test set [6]. The author also develops two reliability models that use execution time and code coverage to analyse the effect of coverage on reliability.

Zhu et al. survey several research studies to examine test adequacy criteria and their role in dynamic testing [41]. Leon et al. empirically compare four techniques for their effectiveness in finding defects: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling [28]. They show that a combination of distribution-based (based on distribution of tests' execution profiles) and coverage-based filtering techniques is effective in prioritizing test cases and reveals more defects than using the either one alone. Andrews et al. use four different types of coverage (Block, Decision, C-Use, and P-Use) and mutants to examine the relationship between test suite size, fault detection and coverage [2]. They show that effectiveness is correlated with all the coverage types. In this study, we analyze a different problem i.e., whether there is a correlation between coverage and the number of bugs found after the release of the software.

Inozemtseva et al. study five large Java systems to analyse the relationship between the size of a test suite, coverage and the test suite's effectiveness [18]. They measure different types of coverage such as decision coverage, statement coverage and modified decision coverage and use mutants to evaluate the test suite effectiveness. The results of their study show that the coverage has a correlation with the effectiveness of a test suite when the test suite's size is ignored, whereas the correlation becomes weak when the size of test suite is

controlled. They also find that the type of coverage has little effect on the strength of correlation. Gopinath et al. analyse thousands of projects from GitHub to identify which coverage criteria is the best estimation of fault detection [13]. They examine tests written by developers as well as tests generated by the automated testing tool Randoop to understand the ability of a test suite to kill mutants. They find that statement coverage is the best coverage criteria to predict the test suite quality. Kochhar et al. study two large open source systems to analyse the relationship of coverage and its effectiveness with real bugs logged in an issue tracking system [25]. They use Randoop, an automatic test-generation tool, to generate test suites on the fixed version and run those suites on the buggy version to analyse the effectiveness of a test suite in killing bugs. They find that coverage is moderately correlated with the effectiveness of a test suite for one project, while strongly correlated for the other one. Namin and Andrews analyze a similar problem on few small systems to see if higher coverage leads to an increase in effectiveness [31]. They find that coverage is related to effectiveness when size is controlled for, whereas size and coverage both used together can lead to better prediction of effectiveness. While the above studies analyse the effectiveness of test suites and coverage in findings bugs, in this study, we analyse the impact of code coverage on the number of real bugs found after the release of the software for large software systems.

Past studies have analysed mutants i.e., artificially injected bugs and their suitability to be used as replacement for real bugs. Andrews et al. use eight well-known C programs and run test cases on real faults and mutants to compare the fault detection ability of test suites on these two versions [3]. They use different mutation operators such as deleting a statement, negating the condition in an if or while statement etc. Their results show that generated mutants are similar to the real faults but different from hand-seeded faults and hand-seeded faults are harder to detect than real faults. In another study, Just et al. study whether mutants are valid substitute for real faults i.e., a test suite's ability to detect mutants is correlated with its ability to detect real faults fixed by developers [20]. They use 5 open-source programs having 357 real faults and find that there is a statistically significant correlation between mutant detection and real fault detection, independent of code coverage. While the above studies show that mutants are representative of real bugs, however, other studies contradict the above argument. Gopinath et al. analyze a large number of projects written in four languages, i.e., C, Java, Python and Haskell [14]. They show that a significant number of changes are larger than the common mutation operators and different languages have different mutation patterns. Namin et al. show that mutation used in testing experiments is highly sensitive to external threats such as test suite size, mutation operators and programming languages [32]. They suggest that generalization of findings based on mutation should be justified by the factors involved.

In a previous study [26], we analyze the correlation between code coverage and several software metrics such as LOC, cyclomatic complexity and number of developers at the project and file level. We find that a large number of projects exhibit low coverage and when the size and complexity increases, coverage decreases at the project level but increases at the file level. In two other studies, we examine the correlation between the number of test cases in a project with several metrics such as programming languages, the number of bugs, the number of bug reporters and the number of developers [22], [23]. To count the number of test cases, we used a heuristic i.e., all the files that contain the "test" in their file name. In this paper, we investigate 100 large open-source projects from GitHub to analyse the impact of *code coverage* on the number of post-release bugs at the project and file level. We use Sonar to calculate the number of test cases and also to run test cases to analyze the impact of coverage on real bugs.

B. Large Scale Studies on GitHub

Jiang et al. collect thousands of forks from GitHub to understand why and how developers fork what from GitHub [19]. They conduct surveys, analyze programming languages and owners of forked repositories. They have several interesting findings a) developers forks repositories to submit pull requests, fix bugs, add nee features etc. and they use various sources such as search engines, external sites (e.g., Twitter, Reddit), social relationships to find repositories to fork, b) developers are more likely to fork repositories written in their preferred language, and c) developers mostly fork repositories from creators. Zhang et al. propose an approach to detect similar repositories on GitHub [40]. They make use of GitHub stars and readme files and use three heuristics: a) repositories with similar readme file content are likely to be similar, b) repositories starred by users having similar interests are likely to be similar, and c) respositories starred within a short period of time are likely to be similar. Based on these heuristics, they build a recommendation system named RepoPal and compare it with state-of-the-art approach CLAN using one thousand repositories on GitHub. Sharma et al. collect 10,000 popular projects on GitHub and propose a cataloging system to group similar projects into categories [33]. They automatically extract descriptive segments from *readme* files and aply LDA-GA, a state-of-the-art topic modeling algorithm that combines Latent Dirichlet Allocation (LDA) and Genetic Algorithm (GA), to identify categories. Their approach can identify new categories to complement exisitng GitHub categories and also identify new projects for existing categories.

Casalnuovo et al. study 69 C and C++ projects to understand the correlation between asserts and defect occurrence and how assertion use is related to ownership and experience of methods by developers [8]. They find that assertions are widely used in these projects and adding asserts has a small yet significant relationship with defect occurrence. They also find that asserts tend to be added to methods with higher ownership and developers with more experience have higher likelihood of adding asserts. Kochhar et al. perform a partial replication of Casalnuovo et al. study [8] to understand the correlation between assert usage and defect occurrence on a large dataset of 185 Java projects from GitHub [24]. They collect several metrics such as number of asserts, number of defects, number of developers and number of lines changed

to a method and also perform an in-depth qualitative study on 575 distinct methods, each containing at least one assert statement to understand assert usage patterns. They find similar results as Casalnuovo et al. that asserts have a small yet significant relationship with defect occcurence. Furthermore, they find that asserts are used for several purposes such as null check, process state check, initialization check, resource check, resource lock check, minimum and maximum value constraint check, collection data and length check and implausible condition check.

Vasilescu et al. analyse 246 projects from GitHub to investigate the impact of usage of Continuous Integration (CI) on quality and productivity [37]. Their results show that teams using CI have more pull requests accepted from core contributors and fewer rejections from external contributors. Gousios et al. analyse pull-based software development model on a dataset on 291 projects from GitHub [15]. They find that only 14% of the active projects use pull-requests and 60% of the pull-requests are processed in a day. Kochhar et al. analyse a large dataset of 628 projects from GitHub to understand the impact of using multiple languages on software quality [27]. They build multiple regression models to study the effect of different languages on the number of bug fixing commits after controlling for factors such as project age, project size, team size, and the number of commits. They find that using multiple languages increases defect proneness and popular languages such as C++, Objective-C, Java etc. are more defect prone when used in multi-language setting. Vasilescu et al. use mixed-methods approach by surveying thousands of developers and analysing thousands of projects to investigate how gender and tenure diversity relate to team productivity and turnover [36].

Different from above studies, we investigate the correlation between code coverage and post-release defects on a dataset of 100 large projects from GitHub. We collect real bugs instead of using artificially injected mutants. We analyse correlation between coverage and defects at the project and file level and employ several statistical measures.

VII. Conclusion and Future Work

Test cases are an integral part of any software project as they allow developers to test their code and improve software quality. Code coverage is an important metric that gives information about how much of the code is not covered by test cases, and thus can be a potential source of bugs. Previous research has focused on the number of mutants identified using code coverage. We have conducted a large-scale study to analyze the code coverage of test cases and studied its correlation with the number of post-release bugs logged in the issue tracking system. We used standard statistical analysis and regression to measure the degree of correlation.

The findings of our study are:

1) At the project level, code coverage has an insignificant correlation to the number of bugs as well as to other metrics such as number of bugs/LOC and number of bugs/complexity found after the release of the software. By categorizing projects based on size and complexity, we observe an insignificant correlation between coverage and other metrics.
2) At the file level, there is no correlation between coverage and metrics such as number of bugs/lines of code, number of bugs/cyclomatic complexity and number of bugs/efferent couplings. Coverage/complexity has no correlation with the number of bugs nor with the number of bugs/LOC. By categorizing files based on size of the project they belong to, we observe no correlation between coverage and other metrics for files in medium sized projects and insignificant correlation for files in small and large projects. For files present in low and high complexity projects, we observe no and insignificant correlation between coverage and various metrics, respectively.

Our findings highlight that although coverage is commonly used as yardstick for test adequacy, their impact should not be overestimated. For most of the settings considered in this work, the relationship between test coverage and post-release bugs are either non-existent or unclear (i.e., statistically insignificant). Designing test cases for the sole purpose of increasing coverage may or may not translate to higher bug finding rate. In the future, we plan to analyse datasets from other open-source platforms to mitigate the external validity threats. Furthermore, we plan to collect a larger dataset of projects having significant representation across low, medium and high coverage levels to investigate the impact of different coverage levels on the number of post-release bugs.

Dataset

Our dataset is publicly available on GitHub: https://github.com/smusis/coverage-defects.

References

[1] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen. Can testedness be effectively measured. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[2] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[4] S. Androutsellis-Theotokis, D. Spinellis, M. Kechagia, and G. Gousios. Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management*, 4(3–4):187–347, 2011.

[5] A. Bachmann and A. Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 62–71, 2010.

[6] X. Cai. *Coverage-based testing strategies and reliability modeling for fault-tolerant software systems*. PhD thesis, The Chinese University of Hong Kong (People's Republic of China), 2006.

[7] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[8] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 755–766, 2015.

[9] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. Applied multiple regression/correlation analysis for the behavioral sciences. *Lawrence Erlbaum*, 2003.

[10] N. E. Fenton and M. Neil. Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, 2000.

[11] G. Gill and C. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering (TSE)*, 17(12):1284–1288, 1991.

[12] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, 1975.

[13] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 72–82, 2014.

[14] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–200, 2014.

[15] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 345–355, 2014.

[16] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering (TSE)*, 26(7):653–661, 2000.

[17] W. G. Hopkins. *A new view of statistics*. Internet Society for Sport Science, 2000.

[18] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 435–445, 2014.

[19] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang. Why and how developers fork what from whom in github. *Empirical Software Engineering (EMSE)*, 22(1):547–578, 2017.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 654–665, 2014.

[21] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[22] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects-a preliminary study on 50, 000 projects. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 353–356, 2013.

[23] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *13th International Conference on Quality Software (QSIC)*, pages 103–112, 2013.

[24] P. S. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 298–307, 2017.

[25] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015.

[26] P. S. Kochhar, F. Thung, D. Lo, and J. L. Lawall. An empirical study on the adequacy of testing in open source projects. In *21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 215–222, 2014.

[27] P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 563–573, 2016.

[28] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, pages 442–453, 2003.

[29] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, SE-2(4):308–320, 1976.

[30] A. Mockus, N. Nagappan, and T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 291–301, 2009.

[31] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–68, 2009.

[32] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[33] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 314–319, 2017.

[34] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[35] D. Spinellis. *Code Quality: The Open Source Perspective (Effective Software Development Series)*. Addison-Wesley Professional, 2006.

[36] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)*, pages 3789–3798, 2015.

[37] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 805–816, 2015.

[38] A. H. Watson, T. J. Mccabe, and D. R. Wallace. Structured testing: A software testing methodology using the cyclomatic complexity metric. In *National Institute of Standards and Technology*, 1996.

[39] F. Zhang, F. Khomh, Y. Zou, and A. Hassan. An empirical study of the effect of file editing patterns on software quality. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 456–465, 2012.

[40] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li, and J. Sun. Detecting similar repositories on GitHub. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23, 2017.

[41] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.