An Empirical Study on the Adequacy of Testing in Open Source Projects

Pavneet Singh Kochhar[1], Ferdian Thung[1], David Lo[1], and Julia Lawall[2]
[1]Singapore Management University, Singapore
[2]Inria/Lip6 Regal, France
{kochharps.2012, ferdiant.2013, davidlo}@smu.edu.sg, julia.lawall@lip6.fr

***Abstract*—During software maintenance, testing is a crucial activity to ensure the quality of code as it evolves over time. With the increasing size and complexity of software, adequate software testing has become increasingly important. Code coverage is an important metric to gauge the effectiveness of test cases and the adequacy of testing. However, what is the coverage level exhibited by large-scale open-source projects? What is the correlation between software metrics and the code coverage of the software?**

In this study, we investigate the state-of-the-practice of testing by measuring code coverage in open-source software projects. We examine over 300 large open-source projects written in Java, to measure the code coverage of their associated test cases. We analyse correlations between code coverage and relevant software metrics such as lines of code, cyclomatic complexity, and number of developers. Our results show that the coverage level decreases with the increase in size and complexity of the software, whereas the number of developers has an insignificant correlation with the code coverage. However, considering individual files, coverage increases with the size and complexity, whereas the number of developers has no correlation with the code coverage. Our results highlight the strengths and weaknesses of testing in open-source projects and make recommendations for future research.

Keywords*—*Code coverage, Software testing, Test adequacy, Software metrics

I. Introduction

Recent years have seen a substantial increase in the importance of open-source software: the Linux operating system is used on platforms ranging from embedded systems to supercomputers, Android has captured a majority share of the smartphone and tablet market, and applications such as Firefox, Eclipse, and LibreOffice are used across the desktop computing spectrum. Furthermore, the open-source developer community is becoming increasingly professionalized, with large organizations investing time, effort and money in open-source development. The increasing reliance on open-source software implies that it is important to investigate potential weaknesses in its development process.

Years of software engineering research and practice have shown that adequate testing is critical to the development of reliable software [1]. Inadequate testing implies that developers are dependent on users to validate the software, which risks alienating the user base, and, increases the difficulty of identifying and resolving bugs. Nevertheless, developing thorough test cases is hard, and is not essential to the process of implementing a given functionality, and thus we conjecture that open-source developers,[1] who may not be forced to do so by an employer, will not write adequate test cases for their code. Indeed, a decade ago a study of open source developers found that over 80% of the open-source developers surveyed admitted that their projects lack testing plans, even though most projects spend nearly 40% of the time in the testing phase [2].

One metric that is commonly used to measure the adequacy of testing is code coverage, that is, a measure of the set of lines of code or code paths that are executed by a set of tests. Quality managers can use coverage information to assess test suites, decide when to stop testing and examine portions of the code that are not covered and thus may contain faults [3]. Judicious use of code coverage can help in finding new defects and increasing the robustness of the software [4]. Furthermore, software cost models based on coverage information can be used to estimate the cost of testing, the cost of removing faults and the potential risk caused by bugs emerging from uncovered code [5]. Measuring coverage alone, however, is not enough to obtain a complete picture of the state of testing in open-source software. To understand, and potentially improve, the state of testing in open-source software, it is necessary to correlate code coverage information with other software metrics that can characterize the software development process, such as lines of code, cyclomatic complexity, and number of developers. These easy-to-collect metrics can help characterize projects in which testing is insufficient, and thus can help developers and managers assess when more testing effort for their software may be required.

In this study, we investigate over 300 open-source Java projects from the GitHub[2] hosting site and Debian[3] Linux distribution. GitHub hosts millions of software projects including some well-known projects, such as Hadoop[4] from Apache and Aether[5] from Eclipse, whereas Debian is a popular Linux distribution and contains projects such as the Jetty[6] server from Eclipse and the Netty[7] client-server framework. We examine the adequacy of testing of the selected projects by analysing the code coverage reported by executing the test cases present in these projects. We also investigate the relationships between various project characteristics and code coverage. The metrics that we use are well-known, and have been used in many past studies to characterize various software projects [6], [7], [8], [9].

[1]In this paper, we define developers as the people who write code as well as those who write test cases.

[2]https://github.com/
[3]https://www.debian.org/
[4]http://hadoop.apache.org/
[5]http://www.eclipse.org/aether/
[6]http://www.eclipse.org/jetty/
[7]http://netty.io/

We examine the following research questions:

RQ1: *What are the coverage levels and test success densities exhibited by different projects?*

RQ2: *What are the correlations between various software metrics and code coverage at the project level?*

RQ3: *What are the correlations between various software metrics and code coverage at the source code file level?*

We find that only 40 out of the 327 projects have coverage levels between 75 and 100 percent. The average and median coverage levels are only 41.96% and 40.30%, respectively. We also find that projects of larger sizes or higher complexities exhibit lower coverage levels. This highlights the need to improve state-of-the-art test generation techniques, e.g., [10], [11], [12], [13], which have often only been demonstrated to work on small to medium size programs. Nevertheless, we do find that developers care about large or complex files: larger or more complex files exhibit higher code coverage. Finally, there is no or insignificant correlation between the number of developers and code coverage.

The contributions of this paper are as follows:

1) We conduct a large-scale study on over 300 open-source software projects with the objective of understanding the state-of-the-practice of testing in open-source projects.
2) We examine the adequacy of testing by executing the test cases that come with these projects and measuring the code coverage. We report some statistics that depict the test adequacy levels across the projects.
3) We investigate the relationships between various project characteristics, measured by various software metrics, and code coverage, at the project and file levels.

The structure of this paper is as follows. In Section II, we briefly describe test adequacy criteria, and some tools and services that we use in collecting information about open-source projects. We elaborate on our empirical study methodology in Section III. In Section IV, we explain the statistical analysis we performed on the data along with our answers for the three research questions. We discuss our findings and describe threats to validity in Section V and Section VI, respectively. Section VII describes related work. We conclude and mention future work in Section VIII.

II. Background

In this section, we discuss test adequacy criteria, and then briefly describe the various tools on which we rely: Sonar, the tool we used for collecting software metrics, GitHub & Debian, the platform and the Linux distribution from which we have collected the projects for our study, respectively, and Maven,[8] the automation tool used to build the projects.

A. Test Adequacy Criteria

Software testing is used to assess the functionalities of a program or system and to investigate whether the tested entity produces the expected results based on a set of inputs and execution contexts. A *test case* is an input on which the program under test is executed during testing. A *test suite* is a set of test cases. A *test adequacy criterion* is a predicate that defines the properties that must be satisfied to constitute a thorough test [14].

Test adequacy is commonly measured in terms of some form of *code coverage*, which is a measurement of the portion of the code that is executed by running test cases. A set of test cases that satisfies the coverage criterion is said to be adequate. Code coverage can be measured in terms of lines, branches etc. Line and branch coverage, which are often used in practice, are defined as follows

Line coverage: Line coverage reports the thoroughness of testing in terms of which lines of code were executed during testing. The associated adequacy criterion requires execution of all the lines. The percentage of executed lines indicates the adequacy of testing.

Branch coverage: Branch coverage reports the thoroughness of testing in terms of the set of branches that were executed during testing. The associated adequacy criterion requires execution of all the branches. The percentage of executed branches indicates the adequacy of testing.

We use Sonar, described below, to collect coverage information. When unit test cases are run, Sonar reports *overall coverage*[9]; overall coverage combines line and branch coverage as follows:

$$Overall\,Coverage = \frac{CT + CF + LC}{2 * B + EL} \quad (1)$$

CT is the number of branches that evaluate to 'true' at least once; *CF* is the number of branches that evaluate to 'false' at least once; *B* is the total number of branches; *EL* (Executable Lines) is the number of lines that could be covered by unit tests, i.e., lines of code, excluding blank and commented lines; *LC* (covered lines) is the difference between *EL* and the number of lines of code not covered by any unit tests.

B. Sonar

Sonar[10] is an open-source platform to manage software quality. It is designed as a web-based application that can be installed standalone or can be integrated into an existing Java Web Application Container such as Tomcat. Sonar integrates seamlessly with build automation tools such as Maven and Ant. It supports analysis of design, architecture and object-oriented metrics for Java applications.

Sonar combines various existing tools, such as JaCoCo,[11] for getting code coverage, and Surefire,[12] for executing unit test cases. Sonar can also compute the various software metrics that we use in this study, such as lines of code, cyclomatic complexity, number of test cases etc. Sonar considers each method in a test case file as one test case. In terms of code coverage, Sonar gives information about line coverage, branch coverage and overall coverage. Thus, Sonar provides a single platform to collect all of the data that we are interested in, which is the reason we pick Sonar for this study.

[8] http://maven.apache.org/

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

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

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

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

C. Maven

Maven is a build automation tool that primarily supports projects implemented in Java. Projects using Maven can be built using information stored in the project object model (POM) file, i.e., *pom.xml*. This file provides information about the project, its dependencies on other modules and plug-ins and the order in which the project's components will be built. Maven dynamically downloads all the dependent plug-ins and Java libraries and adds all these artefacts to the local repository. Sonar leverages Maven's project directory structure to gather information such as the number of lines of code, the number of test cases, the number of packages, and the number of classes and to run test cases to collect the coverage level of the project.

D. GitHub & Debian

GitHub is a web-based project-hosting platform based on the git[13] revision control system. GitHub has become one of the largest collaboration networks of software developers, hosting more than 3,000,000 users and over 5,000,000 repositories. GitHub hosts project repositories ranging from gaming, to operating systems to web servers, written in various languages. This makes it an interesting source on which to conduct an empirical study. We clone the git repositories of projects that use Maven. Based on this criterion, GitHub provided an initial set of 757 projects.

Debian is one of the most popular Linux distributions. It provides a wide range of open-source software, some of which use the Maven build system. We cloned the repositories of projects that use Maven when possible using the URL provided in the Debian metadata. In total, we got 228 projects. These projects use different version control systems including git, svn and mercurial.

We took the union of these projects collected from GitHub and Debian, removing duplicates when projects appear in both sources. Overall, our dataset consists of 945 projects which we use for further analysis.

III. Methodology & Basic Statistics

We now describe how we collect data for the empirical study. We also provide some statistics describing the collected dataset.

A. Methodology

For our empirical study, we downloaded the projects from GitHub and the projects distributed by Debian. Our dataset includes projects developed by well-known organisations such as The Apache Software Foundation and The Eclipse Foundation.

We clone projects that use Maven as the project management tool. Out of 945 projects, 872 projects contain test suites. We analyse these projects and run test cases. For project set-up, we run the command `mvn clean install`, which clears any pre-compiled files of previous builds, builds a dependency tree for all the sub projects specified in the *pom.xml* (the root POM) and compiles all the *.java* files. Then, we run Sonar using the command `mvn sonar:sonar`, which performs dynamic analysis by running test cases and then creates reports based on the results. Unfotunately, many of the projects had compilation errors and dependencies on unavailable external libraries. This observation is consistent with the results of others [15]. We tried to resolve these issues, however, if after some effort the project still failed, we discarded the project. In the end, we have 327 projects that successfully compile, run test cases and produce coverage.

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

B. Statistics

We now present some statistics describing the data collected in our empirical study. We compute these statistics to characterize the projects in our dataset and assess the suitability of these projects as representative samples to answer our three research questions. These basic statistics also describe the range of values of the various metrics for the projects in our dataset. We analyse the correlations of these metrics with code coverage in Section IV.

a) Lines of code (LOC): We used Sonar to count the lines of code of projects in our dataset, excluding comments, blank lines and test cases. Figure 1a depicts the distribution of the number of lines of code of the projects in our dataset. 90 projects have between 1 and 5,000 LOC, 56 projects have between 5,000 and 10,000 LOC, 129 projects have between 10,000 and 50,000 LOC, and 25 projects have between 50,000 and 100,000 LOC and 27 projects have more than 100,000 LOC. The mean size of the projects is 31,120.71 LOC and the median size is 11,484 LOC. The largest project in our dataset is Apache Hadoop, which contains 454,137 LOC.

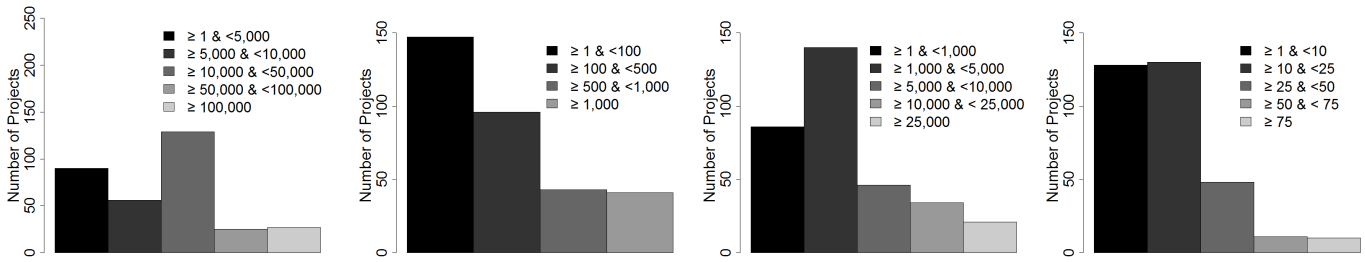
b) Test Cases: We use Sonar to collect the total number of test cases for each project. Sonar also gives information about the number of test cases that failed and the number of test cases that were skipped. Test cases can be skipped due to compilation errors, missing dependencies, etc.

Figure 1b shows the distribution of test cases across projects. 147 projects have fewer than 100 test cases, whereas 41 projects have more than 1,000 test cases. 96 projects have between 100 and 500 test cases and 43 projects have between 500 to 1,000 test cases. The number of test cases varies from 1 to 31,414. The mean number of test cases per project is 563.97 and the median value is 141.

c) Cyclomatic complexity: Cyclomatic complexity is a measure of the number of linearly independent paths through the source code of a software program [16]. Cyclomatic complexity is particularly useful in approximating the number of test cases necessary for independent path testing [17]. A program with low cyclomatic complexity is typically easier to maintain [18].

Figure 1c depicts the distribution of cyclomatic complexity of projects in our dataset. 86 projects have complexity between 1 and 1,000, 140 projects have complexity between 1,000 and 5,000, 46 projects have complexity between 5,000 and 10,000 and 34 projects have complexity between 10,000 and 25,000. 21 projects have complexity above 25,000 with the highest complexity value being 114,045. We can observe that most of the projects have complexity below 10,000.

d) Developer contributions: Our projects use different version control systems such as *git*, *svn* and *hg (mercurial)*, so we use *git log*, *svn log*, and *hg log*, respectively, to examine the commit history of all the projects and to extract the names of all of the developers working on these projects. We also



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

Figure 1: Distribution of Projects

collect developer information at the file level, i.e., the number of developers who have made changes to each file.

Figure 1d shows the distribution of the number developers of all the projects. 128 projects have between 1 and 10 developers, 130 projects have between 10 and 25 developers, 48 projects have between 25 and 50 developers and 11 projects have between 50 and 75 developers. 10 projects have more than 75 developers, with the project *Netty* having the highest number of developers i.e., 146. The mean and median numbers of developers across all the projects are 18.15 and 13, respectively.

IV. Empirical Study Results

In this section, we examine the research questions and report the results of our empirical study.

A. RQ1: What are the coverage levels and test success densities exhibited by different projects?

Motivation: Investigating test cases and coverage level of a project is important in understanding the reliability of the software project. A test suite with high coverage is likely to have a higher fault detection capability and to better help developers find bugs than the one with low coverage [19].

Findings: Table I shows the distribution of coverage levels. Most of the projects exhibit low coverage levels, as the average coverage (i.e., sum of coverage of all projects divided by number of projects) is only 41.96% and the median coverage is only 40.30%. Almost one-third of the projects have coverage between 0% and 25%.

Table I: Project Distribution across Coverage Levels

Coverage Level (%)	Number of Projects
0-25	105
25-50	90
50-75	92
75-100	40

Coverage indicates the amount of code touched by the test cases, but does not ensure that the program runs correctly on the tests. We thus next calculate test success density as the number of test cases that are executed successfully out of the total number of test cases. Figure 2 depicts the test success density of all the projects in our dataset. We observe that 254 projects have test success density greater than or equal to 98%, out of which 200 projects have 100% success density. 45 projects have test success density between 75% and 98%, and 6 projects exhibit success density between 25% and 50%. Only 9 projects in our dataset show a success rate below 25%.

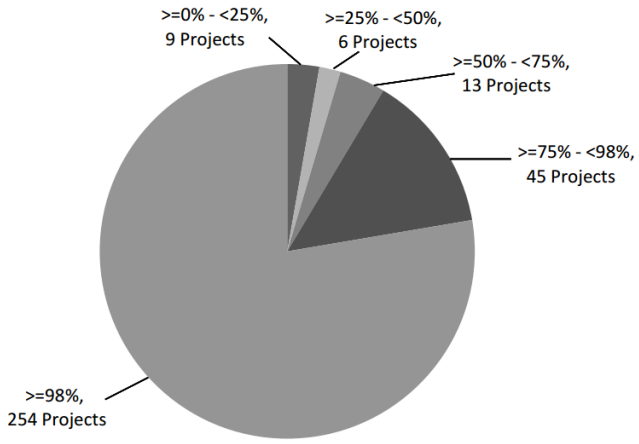


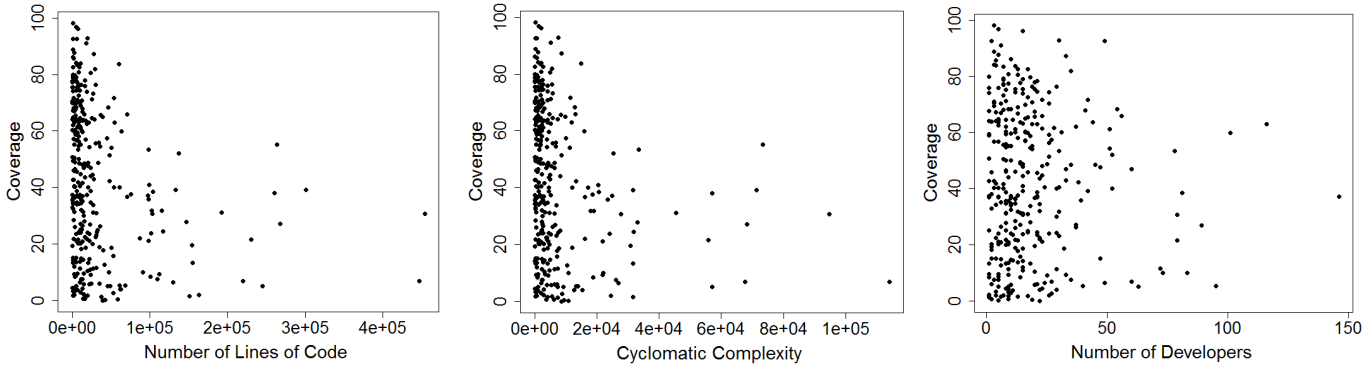
Figure 2: Test Success Density

The levels of coverage varies from 0.1% to 98.2% with an average value of 41.96%. Almost one-third of the projects have coverage levels between 0% to 25% which highlights that projects have low code coverage. 61.16% of the projects have 100% test success density.

B. RQ2: What are the correlations between various software metrics and code coverage at the project level?

In this research question, we examine the correlation between code coverage and various software metrics such as lines of code, complexity, number of developers, age of the project, and popularity of the project.

Motivation: Software metrics, such as lines of code, code coverage, cyclomatic complexity, etc., give quantitative measurements of the degree to which a software project and its development process exhibit a particular attribute. These metrics can be used to improve the software quality, to analyse the productivity of a software project team, to anticipate the future needs of developers and to estimate the amount of maintenance required for the project. Comparing various metrics with code coverage can help us understand which project attributes are correlated to the adequacy of testing. This understanding can help us identify characteristics of projects that are prone to inadequate testing.



(a) Number of Lines of Code vs. Coverage (b) Cyclomatic Complexity vs. Coverage (c) Number of Developers vs. Coverage

Figure 3: Scatter Plots (Project Level)

Findings: First, we analyse the correlation between lines of code and amount of code coverage. As the quality of the software is related to coverage [20], we believe that the coverage should either remain the same or increase with an increase in LOC to ensure that new parts are covered by test suites.

The scatter plot (Figure 3a) between the number of lines of code and coverage shows that as the number of lines of code increases, the coverage level actually decreases. The Spearman's ρ for the distribution is -0.306 with p-value = $1.566e^{-08}$, which shows that there is a negative correlation between number of lines of code and code coverage.

Cyclomatic complexity of software generally increases with an increase in the number of lines of code [8], [21]. As the complexity of a software project increases above a threshold, the software becomes error prone [17]. Figure 3b depicts the scatter plot between coverage and cyclomatic complexity. The coverage level decreases with an increase in the complexity of the code. Spearman's ρ for the distribution is -0.276 (p-value = $3.665e^{-07}$), which shows a negative correlation between cyclomatic complexity and code coverage.

The above observations highlight that open-source developers need to increase the testing effort, to maintain or increase the code coverage level with the increase in size or complexity of the software. Cyclomatic complexity has a significant impact on testing [17]. Thus, developers who are working on large and complex projects should put more emphasis on testing to improve the reliability of the software.

Test cases are contributed by the developers of a software project. These developers play a significant role in writing and running these test cases. Figure 3c depicts the scatter plot between the number of developers and the code coverage of the project. We observe from the graph that the coverage slightly decreases when the number of developers increases. The Spearman's ρ value is 0.016, with a p-value of 0.763, which shows that the correlation between the coverage level and the number of developers is insignificant.

The largest projects exhibit lower coverage levels. Also, coverage levels of the projects decrease with the increase in cyclomatic complexity. The number of developers has an insignificant correlation with the coverage of the projects.

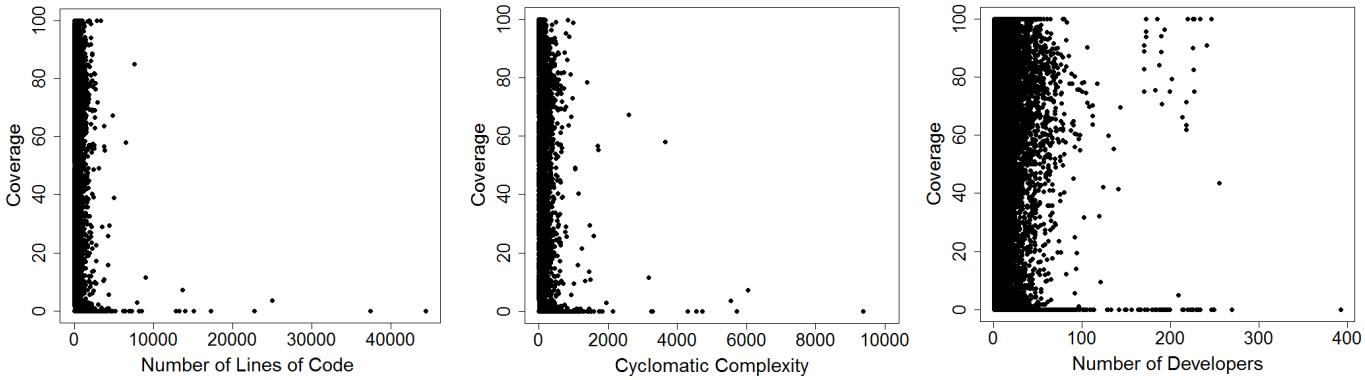
C. RQ3: What are the correlations between various software metrics and code coverage at the source code file level?

Motivation: The coverage level of the overall software gives an idea of how well a project is tested. However, a project may consist of many files having diverse properties. So, we want to additionally examine the software metrics at the file level, which can help us to study how these metrics, which vary from file to file, are correlated to code coverage. This can enhance our understanding of the characteristics of files that are inadequately tested.

Findings: We extract the number of lines of code and coverage level for all of the files that constitute a project. In total, we have 107,762 Java class files accumulated over all the projects. Figure 4a shows the scatter plot of the number of lines of code and coverage. The results are contrary to the correlation between LOC and coverage at project level (Figure 3a). The Spearman's ρ for the distribution is 0.180 (p-value $< 2.2e^{-16}$) depicting a small positive correlation.

We proceed to investigate the correlation between complexity and coverage at the file level. In Figure 3b we observed that with an increase in the complexity of the system, the coverage of the system drops. We want to determine if more complex files are less covered than less complex files, which would lead to an overall reduction in the coverage of the software. We draw a scatter plot depicting the relationship between complexity and coverage level of source code files in Figure 4b. The Spearman's ρ for the distribution is 0.221 (p-value $< 2.2e^{-16}$), which shows that there is small positive correlation between cyclomatic complexity and code coverage. The results are contrary to the correlation of complexity and coverage for the overall project.

Finally, we examine the correlation between the number of developers and the coverage levels of files created by those developers. For each file, we consider the number of developers to be the number of people who have been the author of at least one commit that touches the file. Figure 4c depicts the correlation between the number of developers and coverage for all the files contained in the projects. There is no correlation between the number of developers and coverage level of the files. The Spearman's ρ value is 0.050 with p-value $< 2.2e^{-16}$.



(a) Number of Lines of Code vs. Coverage (b) Cyclomatic Complexity vs. Coverage (c) Number of Developers vs. Coverage

Figure 4: Scatter Plots (File Level)

At the file level, the results are contrary to the results at the project level. Coverage level increases with the increase in size as well as complexity of the files. However, there is no correlation between the number of developers and coverage at the file level.

V. Discussion

Despite the benefits of software testing, our study finds that most open-source projects are poorly tested – the median coverage across the 327 projects is only 40.30%. Our study is the first to demonstrate this phenomenon over large software projects. Our results suggest that open-source developers pay little attention to testing. This is undesirable as open-source software development is arguably as important as its closed-source counterpart. This is supported by the fact that many real systems are built on top of open-source projects and libraries, e.g., hadoop, struts, etc. Hence, the quality of many open-source projects is important, as bugs and failures can adversely affect many users. Furthermore, a past study has indicated that software quality is correlated to coverage [20]. Therefore, there is a need to further improve software testing, in general, and test coverage, in particular, in open-source projects. This highlights the need for additional research in the software engineering research community to promote new tools that can be easily used by these developers so that the quality of software testing can improve for many open-source systems.

From the results of RQ2, we observe that larger and more complex projects are likely to have *lower* test coverage levels. This indicates that it is harder for developers to keep up with software testing efforts for larger or more complex projects. Indeed, for larger projects, there are more lines to cover; for more complex projects, it is harder to achieve a higher coverage when test cases are added. Thus, there is a need for more help to cover larger software systems. There have been a number of studies on test generation techniques which can help developers achieve a higher coverage during testing [10], [11], [12], [13]. However, most of these studies, have often only been applied to very small (<1 kLOC), small (<10 kLOC), or medium sized programs (<100 kLOC). Thus, there is a need to develop test generation techniques that could work on large projects as these projects would benefit most from test generation techniques.

Our experiments also shed light on the behaviours of software developers in testing their projects. From the results of RQ3, we find that larger and more complex *files* are likely to have *higher* test coverage levels. This indicates that developers pay more attention to files that are large or complex. Research has shown that such files are more likely to be buggy [22]. We also find that the correlation between the number of developers and coverage is insignificant. This indicates that adding more developers to a project does not necessarily increase coverage – the new developers can help to develop more features rather than focus on testing efforts.

VI. Threats to Validity

As with any empirical study, we also have several threats to validity. Some of these threats are described below:

Threats to external validity relate to the generalizability of our results. We have investigated over 300 open-source projects of substantial size from GitHub, which is one of the most popular super-repositories, and from Debian, which is a popular Linux distribution. Projects in GitHub are not restricted to a particular domain. Debian also includes projects from different domains. Thus, we believe there is little threat by only analysing these projects. All the projects that we analyse are written in Java. It is unclear if the results would be the same for projects written in other programming languages. In the future, we plan to reduce these threats to external validity by extending our study to also analyse projects written in various programming languages.

Threats to internal validity relate to the conditions under which the experiments are performed. We use Sonar to find software metrics such as lines of code, number of tests, cyclomatic complexity and code coverage. Sonar relies on Maven, implying that non-Maven projects are not taken into account, and that projects that do not fully conform to the Maven directory structure may be interpreted incorrectly. The latter issue could lead to incorrect calculations of some software metrics such as lines of code. Similarly, during dynamic analysis, i.e., running of test cases, there may be cases where Sonar is not able to find and run all of the test cases, leading to a wrong coverage value. We have, however, manually checked some of the projects, and these appear to fully conform to Maven directory structure. For all the projects in our dataset, we use the latest revision of the repository, rather than a released

version. We do so to check whether project is regularly tested by the developers before it is available to the users. We calculate the number of developers for a file as the number of people who have committed at least one change that modifies the file. It is possible that a developer only makes non-essential changes to a file; still we include that developer. We also only run the Junit tests that Sonar identifies; there could be some test files that are missed by Sonar and these might impact the coverage levels.

Threats to construct validity relate to the suitability of the metrics that we investigate in this study. We use standard metrics, e.g., code coverage, cyclomatic complexity, lines of code, etc. These metrics have often been used to characterize software systems [6], [16].

VII. Related Work

Here, we describe past studies on testing, code coverage and GitHub. Our survey here is by no means complete.

A. Studies on Testing & Code Coverage

Numerous studies have investigated the importance of testing and code coverage on the overall quality and reliability of the software. Mockus et al. study two industrial software projects (Windows Vista from Microsoft and an application from Avaya) to understand the importance of test coverage on test effectiveness and analyse the required test effort with different levels of test coverage [20]. The results of their study show that increased coverage leads to a reduction in the number of post-release defects but increases the amount of test effort. Cai and Lyu study the relationship between code coverage and fault detection capability using coverage testing and mutation testing [23]. They analyse a large project to understand the relationship under different conditions of coverage metrics and testing profiles. Cai performs an empirical investigation to find the effect of code coverage on fault detection and finds that this effect varies under different testing profiles [24]. The results show that there is strong correlation between code coverage and fault detection for exceptional test cases. Shamasunder perform an empirical study to analyse the impact of different kinds of coverage on test suite effectiveness [25]. The results show that branch and block coverage have a higher correlation with fault detection as compared to path coverage. The above studies show the need for achieving good test coverage to improve the reliability of systems. In this work, we investigate the state-of-the-practice of testing by analysing over 300 projects. We measure code coverage of the projects that successfully compile and analyse the relationships between various project characteristics (e.g., size, complexity, etc.) with code coverage. Our study shows that despite the importance of testing, for many projects, the code coverage is low.

Gopinath et al. investigate the correlation between test suite coverage and its effectiveness in killing mutants [15]. They start with more than 1,000 GitHub projects but need to remove most of them due to compilation, etc. errors. They end up with around 200 GitHub projects for their analysis. Most of the projects analysed are small (less than 1000 lines of code). They find that there is a correlation between test suite coverage and effectiveness. Our work complements this work by addressing a different set of research questions. We also study a larger set of projects and most of them are of larger sizes (more than 10,000 lines of code). Inozemtseva and Holmes also investigate the correlation between test suite coverage and its effectiveness in killing mutants on 5 large Java programs [26]. They find that there is a weak to moderate correlation between test suite coverage and its effectiveness. Our work complements this work by addressing a different set of research questions. We also study a large set of projects instead of only 5 projects. Many of the projects that we analyse are as big as the projects that are analysed by Inozemtseva and Holmes (more than 100,000 lines of code).

Several studies have proposed new techniques and methods to increase code coverage. Thummalapenta et al. develop an approach that takes as input a user-specified intent and produces programs in the form of method sequences to produce the object state specified by user [27]. Their approach uses data from static as well as dynamic analysis and is able to produce higher coverage than existing approaches. Pandita et al. propose an approach to produce test inputs to achieve logical coverage and boundary-value coverage using existing test-generation approaches [28]. Their approach is able to increase the coverage and improves the fault detection capability of new test cases. Park et al. propose a new approach, which combines random testing with techniques such as static program analysis and concolic execution [29]. They tested their approach on twelve Java applications and their results show that their approach performs better than some previous approaches such as pure random, adaptive random, and Directed Automated Random Testing (DART). Our work provides added motivation to studies on automatic test generation, including the above mentioned work. We notice that in practice, for many projects, open-source developers do not test their code sufficiently. We also highlight a need to make automated test generation techniques more scalable since coverage tends to go down when project size increases. Thus, large projects are the ones that would benefit more from automated test generation techniques.

In our previous paper, we investigated the correlation of the number of *test files* with the number of developers, the number of bugs, the number of bug reporters and programming languages [30]. We used a heuristic, i.e., treat all the files whose names contain the word *test* as test files, to identify files that contain test cases. In this project, we use Sonar to identify and run test cases. We study *code coverage* rather than the number of test files.

B. Studies on GitHub

In this work, we investigate over 300 projects from GitHub and Debian. Many other studies also investigate GitHub albeit for different purposes. Dabbish et al. examine the value of transparency of large-scale distributed collaborations and communities on GitHub [31]. They interviewed several GitHub users to understand how a user infers the technical goals and vision of others based on the code they edit. Pham et al. conduct interviews with several users of GitHub to understand the impact of transparency on the testing culture [32]. They present several strategies and challenges faced by collaborating developers and project managers. Further, they give suggestions to mitigate these risks to improve the testing behavior of people working on the same project. Bissyandé et al. investigate the

popularity of different programming languages used in tens of thousands of projects hosted in GitHub [33]. Thung et al. investigate the network structure of projects hosted on GitHub [34]. They show that social coding enhances the collaboration among open-source developers. Jing et al. analyse many GitHub repositories and users that participate in them to study project dissemination characteristics [35]. They find that social links play an important role in project dissemination.

VIII. Conclusion and Future Work

During software maintenance, testing is a crucial activity to ensure the reliability of systems. Many past studies have shown that test coverage has an effect on the quality of software systems. In this work, we investigate the state-of-the-practice of testing in the open source community. We examine over 900 open-source projects and investigate the relationships between project characteristics, measured by various software metrics, and code coverage for 327 projects which sucessfully compile and produce coverage.

Our empirical study highlights the following results:

1) Most of the projects have low coverage levels, with an average of 41.96%.
2) 254 out of the 327 projects that build successfully have test success density above 98%.
3) Code coverage of a project decreases with the increase in the size as well as cyclomatic complexity of the project.
4) However, at the file level, coverage increases with the increase in the size of the file as well as its complexity.
5) The number of developers has an insignificant correlation with the coverage at the project level and no correlation with the coverage at the file level.

Our results suggest that many open source developers do not test their code sufficiently. There is a need for additional tools and techniques to help developers increase coverage, especially for large or complex projects. Unfortunately, many current test generation techniques only work on small or medium size projects. Our results also highlight the behaviours of software developers in testing their projects: developers are often aware of the risk of large or complex source code files and put more effort into testing such files, and adding developers does not necessarily increase coverage.

In the future, we plan to expand our study to include more projects to mitigate the threats to external validity. Furthermore, we intend to include other software metrics such as number of defects. As a large number of projects show low coverage, we also plan to analyse the amount of effort required to attain a particular level of coverage.

References

[1] J. R. Horgan and A. P. Mathur, "Software testing and reliability." McGraw-Hill, Inc., 1996.

[2] L. Zhao and S. Elbaum, "A survey on quality related activities in open source," *ACM SIGSOFT Software Engineering Notes*, pp. 54–57, 2000.

[3] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[4] S. Berner, R. Weber, and R. K. Keller, "Enhancing software testing by judicious use of code coverage information," in *ICSE*, 2007, pp. 612–620.

[5] H. Pham and X. Zhang, "NHPP software reliability and cost models with testing coverage," *European Journal of Operational Research*, pp. 443–454, 2003.

[6] N. E. Fenton and M. Neil, "Software metrics: roadmap," in *ICSE*, 2000, pp. 357–370.

[7] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005, pp. 284–292.

[8] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, pp. 797–814, 2000.

[9] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *ESEC/FSE*, 2009, pp. 91–100.

[10] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *ISSTA*, 2010, pp. 183–194.

[11] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for C/C++ using concolic execution," in *ICSE*, 2013, pp. 132–141.

[12] D. Cotroneo, R. Pietrantuono, and S. Russo, "A learning-based method for combining testing techniques," in *ICSE*, 2013, pp. 142–151.

[13] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *ICSE*, 2013, pp. 162–171.

[14] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493–510.

[15] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *ICSE*, 2014, pp. 72–82.

[16] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, pp. 308–320, 1976.

[17] 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.

[18] G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering*, pp. 1284–1288, 1991.

[19] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE*, 1994, pp. 191–200.

[20] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *ESEM*, 2009.

[21] Y. W. Kim, "Efficient use of code coverage in large-scale software development," in *CASCON*, 2003, pp. 145–155.

[22] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012, pp. 14–24.

[23] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, 2005, pp. 1–7.

[24] X. Cai, "Coverage-based testing strategies and reliability modeling for fault-tolerant software systems," Ph.D. dissertation, 2006.

[25] S. Shamasunder, "Empirical study - pairwise prediction of fault based on coverage," Master's thesis, 2012.

[26] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *ICSE*, 2014, pp. 435–445.

[27] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *OOPSLA*, 2011, pp. 189–206.

[28] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *ICSM*, 2010, pp. 1–10.

[29] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *FSE*, 2012, pp. 35:1–35:11.

[30] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *QSIC*, 2013, pp. 103–112.

[31] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *CSCW*, 2012, pp. 1277–1286.

[32] R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *ICSE*, 2013, pp. 112–121.

[33] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *COMPSAC*, 2013, pp. 303–312.

[34] F. Thung, T. F. Bissyandé, D. Lo, and L. Jiang, "Network structure of social coding in GitHub," in *CSMR*, 2013, pp. 323–326.

[35] J. Jiang, L. Zhang, and L. Li, "Understanding project dissemination on a social coding site," in *WCRE*, 2013, pp. 132–141.