How Effective is Continuous Integration in Indicating Single-Statement Bugs?

Jasmine Latendresse*, Rabe Abdalkareem**, Diego Elias Costa*, and Emad Shihab*
*Concordia University, Montreal, Canada
Email: {j_latend,diego.costa,eshihab}@encs.concordia.ca
**Queen's University, Kingston, Canada
Email: abdrabe@gmail.com

Abstract—**Continuous Integration (CI) is the process of automatically compiling, building, and testing code changes in the hope of catching bugs as they are introduced into the code base. With bug fixing being a core and increasingly costly task in software development, the community has adopted CI to mitigate this issue and improve the quality of their software products. Bug fixing is a core task in software development and becomes increasingly costly over time. However, little is known about how effective CI is at detecting simple, single-statement bugs.**

In this paper, we analyze the effectiveness of CI in 14 popular open source Java-based projects to warn about 318 single-statement bugs (SStuBs). We analyze the build status at the commits that introduce SStuBs and before the SStuBs were fixed. We then investigate how often CI indicates the presence of these bugs, through test failure. Our results show that only 2% of the commits that introduced SStuBs have builds with failed tests and 7.5% of builds before the fix reported test failures. Upon close manual inspection, we found that none of the failed builds actually captured SStuBs, indicating that CI is not the right medium to capture the SStuBs we studied. Our results suggest that developers should not rely on CI to catch SStuBs or increase their CI pipeline coverage to detect single-statement bugs.

I. Introduction

Continuous integration (CI) is commonly used in many industry and open-source projects [1]–[3]. Online CI services, such as Travis CI, continuously integrate code changes by automating compilation, building, and testing [3], [4]. With CI, incremental changes brought to the code base are more atomic, which makes bug detection simpler and quicker. In addition, early bug detection and reporting significantly reduce maintenance overhead since it allows developers to fix faults and make possible critical decisions earlier in the project's lifecycle, which leads to fewer unintended consequences [1], [5], [6].

Recently, Karampatsis and Sutton distinguished a new type of software bugs called single-statement bugs (SStuBs) [7]. SStuBs are bugs in which the associated fixing commit contains only single-statement changes, excluding stylistic changes and differences in comments. At first glance, SStuBs tend to be easy to introduce because they can be caused by simple modifications such as changing a variable name or arguments in a function. However, SStuBs still find their way to software projects, even in the presence of a CI pipeline. While prior work on CI focused on studying its usage and benefits (e.g,. [1], [8]) and examining the reasons for failing builds (e.g,. [1], [5]), no prior work answers the question; *how effective is CI in indicating single-statement bugs (SStuBs)?*

Therefore, the main goal of our work is to empirically investigate the effectiveness of CI in identifying and reporting SStuBs, through failing tests. We begin by examining the ManySStuBs4J dataset [7] and selecting 14 open-source Java projects that contain a significant number of single-statement bugs. We then analyze the selected projects to identify the commits that introduce SStuBs in these projects. Finally, we link these commits to their build results on Travis CI [9] to examine how effective is CI in identifying SStuBs. We formulate our study in the follow two research questions:

RQ1: How many CI builds fail when the SStuBs are introduced? How many CI builds fail just before SStuBs are fixed? We find that only 2% of the commits that introduced SStuBs have builds that report a failure. Similarly, only 7.5% of commits preceding the SStuBs fix commit show any signs of build failure. In fact, the majority of SStuBs (50.5%) we investigate have a long time-span, living in the code for more than one month.

RQ2: From the CI builds that do fail, how many fail due to the SStuBs? Of the 23 failed builds that we manually inspected to determine the failure root cause, none failed due to tests covering SStuBs. Instead, builds failed for external reasons, such as dependency errors [10].

Our results show that CI is not effective in capturing SStuBs, hence, developers should not depend on it for such.

II. Case Study Design

The goal of our study is to investigate the effectiveness of CI on identifying SStuBs. To that aim, we first identify the commits that introduce these SStuBs (i.e,. bug-inducing commits) and the commits that precede the SStuBs' fix. Then, we examine the CI build results that were triggered by these commits. To do so, we triangulated three different data sets. First, we use the ManySStuBs4J to identify commits that fix SStuBs [7]. Then, we generate a dataset using the Commit Guru tool to identify commits that introduce SStuBs [11]. Finally, we use the TravisTorrent to extract the build results [9]. In the following sections, we discuss the steps used to filter a set of 14 open source projects from the ManySStuBs4J dataset and the methodology used to address our research questions. An overview of our approach is shown in Figure 1.

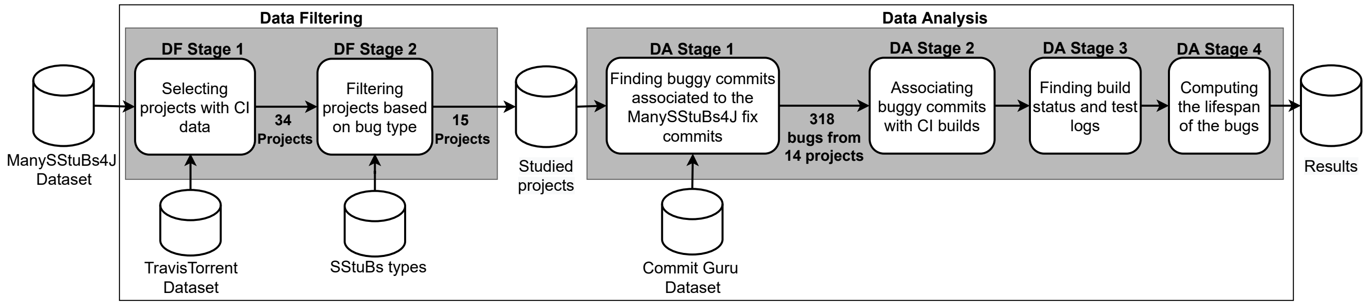
Data Filtering
Data Analysis
DF Stage 1
Selecting projects with CI data
34 Projects
DF Stage 2
Filtering projects based on bug type
15 Projects
DA Stage 1
Finding buggy commits associated to the ManySStuBs4J fix commits
318 bugs from 14 projects
DA Stage 2
Associating buggy commits with CI builds
DA Stage 3
Finding build status and test logs
DA Stage 4
Computing the lifespan of the bugs
ManySStuBs4J Dataset
TravisTorrent Dataset
SStuBs types
Studied projects
Commit Guru Dataset
Results

Figure 1: Overview of our approach.

Table I: Descriptive statistics of the 14 selected Java projects.

Descriptive Statistics	Avg	Min	Median	Max
Project Age (years)	11	8	10	20
KLOC	434	30	380	1,898
# of commits	9,670	1,700	9,863	25,448
# of stars	11,307	5,600	9,600	40,100
Travis CI Usage (years)	6.6	5	7	10
# Bug types	10.6	7	10	15

A. Data Filtering

The ManySStuBs4J dataset [7] is composed of fixes to Java simple bugs. In this paper, we study the small version of the dataset that contains 25,539 SStuBs fixes mined from 100 popular open-source Java projects. While 100 projects were reported by the ManySStuBs4J dataset, we only found 84 projects in the set. Because of time and resource limitations, we decide to filter projects based on the availability of CI data and diversity of bug types, which results in a studied system of 14 Java projects. Next, we describe our filtering steps.

DF 1: Selecting projects with CI data. The goal of our study is to investigate the effectiveness of CI at indicating SStuBs. Hence, we must select projects with publicly available CI data. We select projects that have adopted Travis CI, a popular CI service provider [12]. For this, we cross-reference the TravisTorrent [9], a public dataset of TravisCI data, and the ManySStuBs4J dataset based on the repository names. The repository name is composed by the owner id and the project name, and is unique in both datasets. We find that 34 projects from the ManySStuBs4J dataset have available CI data in TravisTorrent.

DF 2: Filtering projects based on bug type. The ManySStuBs4J dataset presents 16 distinct bug types.

To cover a wider variety of SStuBs categories, we decide to select the 15 projects with the highest diversity in bug types. This resulted in a final set of Java projects containing both SStuBs data and their CI information for a total of 1,284 bug fix commits. Table I presents the descriptive statistics of the selected projects, showing that the projects we investigate are mature (median of 10 years) and popular Java projects. Some of the selected projects include Junit4, Apache Flink, and Google Guice.

B. Data Analysis

Our study focuses on analyzing the effectiveness of CI at indicating the bug, either when it was first introduced (the earliest chance for CI to capture the SStuB) or right before the fix (the latest chance for CI to capture the SStuB). The ManySStuBs4J dataset only contains the commits that introduced the fixes. Hence, to address our research questions, we need data about the commits that introduced the bugs fixed by the commits presented in the ManySStuBs4J dataset. The analysis steps are described below.

DA 1: Finding the bug-inducing commits associated to the ManySStuBs4J fix commits. For this, we use the tool Commit Guru [11] to trace back the bug-inducing commit with the fix commit hash. Commit Guru is a tool that, among other, implements the SZZ algorithm to identify commits that are more likely to introduce bugs into a project. The repository Google/guava failed to be analyzed by Commit Guru due to a faulty commit modifying all repository files. This leaves us with a set of 14 projects for our analysis. Then, we map the corrective commit to the commit it is fixing which corresponds to the bug-inducing commit. We find 318 distinct SStuBs commit and fix commit pairings.

DA 2: Associating buggy commits with CI builds. In this step, we want to obtain CI data for the builds at bug introduction. In other words, we need to see if bug-inducing commits either triggered a build or are part of a push that triggered a build. For this, we query the TravisTorrent dataset for the bug-inducing commit hash. We find that 50 bugs are associated to a Travis CI build.

DA 3: Finding build status and test logs. In this step, we are interested in obtaining the build status and test logs for the builds that are associated to the aforementioned bug-inducing commits. Precisely, we want to obtain the build status and test logs for (a) builds associated with the bug-inducing commits, (b) builds preceding the build associated with the fix commits, and (c) builds associated with the commits that introduced the fix as illustrated in Figure 2. For phase (a), we proceed as mentioned in DA 1. For phase (b), we query the TravisTorrent dataset for the build that corresponds to the previous build associated to a fix commit hash. We obtain 57 builds that occurred because the fix is introduced. For phase (c), we query the TravisTorrent dataset for the fix commit hash and find 366 associated builds. The results found with phase (c) are used to compare the number of tests that run within the CI builds after the fix commit with phase (b).

DA 4: Computing the lifespan of the bugs. To determine

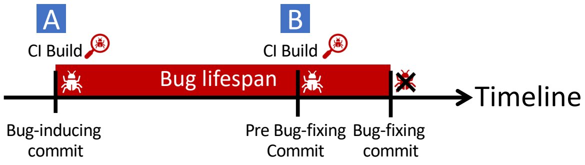

Figure 2: Illustration of our analysis method.

how long a SStuB remains in the code base, we query the Commit Guru set for the commit hash associated to the bug introduction and the author date associated to that commit. We repeat this process for the commit hash associated to the fix. Finally, we calculate the difference in author dates (between introduction and fix) to determine how long the SStuB lived in the code. Using this process, we are able to determine the lifespan for a total of 318 SStuBs.

III. Case Study Results

In this section, we present the results of our two research questions. For each research question, we present its motivation, the approach to answer the question, and the results.

RQ1: How many CI builds fail when the SStuBs are introduced? How many CI builds fail just before SStuBs are fixed?

Motivation: Prior research shows that CI is effective at catching bugs (e.g,. [1], [2]). While easy to fix, SStuBs can linger in the code for quite some time if not captured by automated tests. In this research question, we want to evaluate the CI effectiveness at catching SSTuBs to help developers fix them as soon as possible.

Approach: CI can help identify bugs when they are first introduced (ideal case), or later in the development, once new tests are added to the CI test suite. We illustrate this timeline in Figure 2. We approach this problem by analysing the related build status in two points: at the time when the bug was introduced in the code (stage A in the figure) and right before the bug was fixed by developers, by analyzing the build status related to the commit that preceded the fix-commit (stage B). The rationale is that, the bug introducing commit is the earliest that CI can indicate the presence of SStuBs and the commit that precedes the fix is the last chance for CI to indicate the presence of SStuBs. Then, we report how many commits have triggered the CI to fail, in stages A and B, by computing the proportion of builds that have finished with the status "passed", "failed", and "errored".

While we are interested in evaluating CI effectiveness at capturing SStuBs, CI builds occur at the commit level. A single commit can introduce and/or fix multiple SStuBs, and we find that the 318 SStuBs were introduced and fixed by 240 distinct commit pairs. For instance, we find that the 65 commits introduced 2 different SStuBs in our dataset. That means that all our analysis of CI is based on these 240 distinct commits, as a CI build runs at the commit level.

Results: Of the 240 SStuB related builds, only 2.0% (5) fail when the SStuB is introduced. Table II shows the results of our analysis. In the column "Bug-inducing commits (A)", we show the CI build statuses. From the 64 bug-inducing commits that have their code tested by the projects' test suite, we find that 41 pass all their tests and only 5 (2%) of them have some failed tests. In this result, we also notice that 18 builds yield an error, which can be caused by a myriad of reasons, such as project build errors, server timeout, failure in the environment, etc.

Table II: Coverage of CI on 318 single-statement bugs that were introduced and fixed on 240 distinct commits.

	Bug-inducing commits (A)		Pre Bug-fixing commit (B)	
# CI builds	64	(26.6%)	159	(66.2%)
# CI passed builds	41	(17.0%)	127	(52.9%)
# CI failed builds	5	(2.0%)	18	(7.5%)
# CI builds with errors	18	(5.6%)	14	(5.8%)

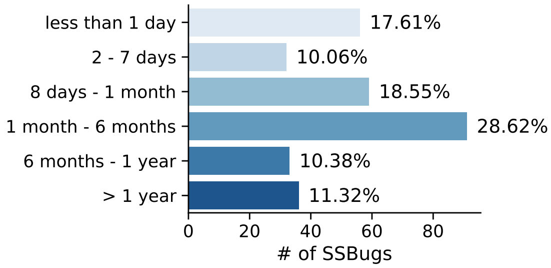

Figure 3: Distribution of the lifespan of 318 SSTuBs in our dataset.

Moreover, we notice that of the 240 bug-inducing commits, only 64 (26.6%) have a CI build data associated with them. This indicates that the vast majority of 176 bug-inducing commits are not tested the projects' CI pipeline.

Of the 240 SStuB related builds, only 7.5% (18) fail just before the fixing commit. "Column Pre Bug-fixing commits (B)" in Table II shows the results of the CI build status just before the fix. From the Table, we notice that the proportion of commits that have triggered a build substantially increases to 159 (66.2%). From these 159 builds, we find that its vast majority (127) pass all their tests. Only 18 builds show failed tests, and 14 builds yield errors.

Overall, only a minute fraction of SStuBs have an associated CI build with failed tests. While the proportion at the time of the fix (stage B) is higher than when introducing the bug (stage A), our results indicate that SStuBs can live in the code base without affecting the build status of the CI pipeline. To investigate how long each SStuBs have lived in the project's code, we compute the lifespan of each bug. Figure 3 presents the distribution of the SStuBs lifespan binned by six categories of periods. Note that the majority of SStuBs (50.32%) have a lifespan longer than one month.

Implications: Surprisingly, the majority of the builds do not fail when SStuBs were introduced to the code base, neither at the builds preceding the fix commit. In fact, most of the studied SStuBs stayed hidden in the code for more than a month, with 22% of them staying in the code for at least 6 months. This indicates some level of inadequacy of CI pipelines in finding

Table III: Why does CI build fail on 23 builds.

	Bug-inducing commits (A)	**Pre Bug-fixing commit (B)**
Total of CI fails	5	18
Failed tests related to SStuBs	0	0
Failed tests unrelated to SStuBs	1	8
Build failed without running tests	3	5
Build failed with all passing tests	1	5

this type of bug. One reason for the the surprisingly long lifespan of these 318 SStuBs is that these bugs may introduce failures in non-essential parts of the software project. Another explanation, is that such bugs were not initially bugs when the code was first modified (in the so-called bug-inducing commits), but they later became bugs in the system due to some other concurrent code change.

> The majority of bug-inducing commits (74%) did not trigger any CI build, and only 5 out of 64 builds failed the test when SStuBs were introduced in the code. Builds that precede the bug fix are more frequent (66%), but only 18 out of 159 builds showed any failed tests. The majority of SStuBs stay in the code for more than a month.

RQ2: From the CI builds that do fail, how many fail due to the SStuBs?

Motivation: The observations made in RQ1 suggest that CI is not very effective in detecting SStuBs early on. In this research question, we are interested in finding how many of CI builds actually captured the SStuBs.

Approach: In this question, we focus on manually analysing the builds that have reported failed tests in RQ1. We analyze the 5 builds that have failed in phase A and the 18 builds that have failed in phase B, as illustrated in Figure 2. We manually inspect the test results and test logs provided by Travis-CI data. Once we identify the failed tests, we resort to code analysis to investigate if the tests failures are linked to the part of the code SStuBs is located. For failed builds, we manually inspect the build logs and test log to verify their relationship. It is important to note that for this step, we do not apply formal analysis since identifying whether the build is related to the test is straightforward.

Results: None of the CI builds that fail, do so because of SStuBs. Table III shows the results of RQ2 in the column "Pre bug-fixing commit (B)". We observe that of 23 builds that failed in our previous analysis, none were associated with the SStuBs in the code. From the 5 builds that fail during bug-inducing commits (A), 1 build fails due to tests completely unrelated to the SStuBs, 3 other builds fail without even running tests and 1 build fails due to external reasons. For example, upon manual analysis of the test logs and source code, we find a failing test associated to the following message "Failure Encountered too many errors talking to a worker node", which is unrelated to the related SStuB.

From the 18 builds that fail right before the bug-fix commit (B), we find that 8 builds have failed tests. However, none of the failed tests execute the code where the SStuBs are located. Furthermore, 5 builds fail without running any tests, due to external reasons such as dependency errors[1] and 5 other builds fail even with all passing tests also because of dependency errors[2].

Implications: Our results indicate that CI are not effective in finding any of the 318 SStuBs we investigated. We find that, while 9 of the 23 builds failed due to failing tests, the tests did not cover the SStuBs code location. Moreover, the majority of the failures were not caused by test suite but external failures such as dependency errors. Hence, our observations suggest that it is more common for builds to fail because of external failures than SStuBs.

> From the 23 failed builds, none are caused by a test affected by SStuBs. Most of the failed builds (14 out of 23) are caused by external failures, such as dependency errors, and did not even execute the test suite.

IV. Threats to Validity

There are few important limitations to our work that need to be considered when interpreting our findings. First, to identify the commits that introduce the SStuBs, we use the Commit Guru tool which is based on the SZZ algorithm [11]. Hence, we are limited by the accuracy of Commit Guru. In some cases, we may have missed instances of commits. To help alleviate this issue, we manually investigated some of the identified commits, and in all cases, we found that Commit Guru has identified the correct commits. Second, our study focuses on only 14 open source Java based projects on a subset of SStuBs and are unlikely to generalize beyond this set of projects. Finally, our study uses the TravisTorrent dataset, thus, our results are limited to the correctness and quality of the available Travis CI build data.

V. Conclusions

In this paper, we analyze the effectiveness of CI on 14 popular open source Java-based projects to warn about single-statement bugs. To do so, we analyze the status of CI builds in two stages: when the SStuBs was introduced in the code and before the SStuBs were fixed. We then investigate how often CI indicates the presence of 318 SStuBs, through test failure. Our findings show that CI was ineffective at indicating the presence of SStuBs, with no build failure caused by SStuBs. In fact, the majority of the studied SStuBs stay in the code for more than a month, which further corroborates with our assessment on the CI inadequacy in capturing SStuBs. These results should, however, be considered preliminary. Future work should focus on enlarging our dataset and examining other programming languages and CI services. Another direction could be to collect qualitative data from projects to assess the quality of the CI pipelines. Interesting future work is to build tools that determine the effectiveness of the CI pipeline.

[1] https://travis-ci.org/github/graylog2/graylog2-server/builds/31332558

[2] https://travis-ci.org/github/prestodb/presto/builds/36184662

References

[1] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 426–437.

[2] 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*, ser. FSE '15, ACM, 2015, pp. 805–816.

[3] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, IEEE Press, 2017, pp. 356–367.

[4] R. Abdalkareem, S. Mujahid, and E. Shihab, "A machine learning approach to improve the detection of ci skip commits," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. DOI: 10.1109/TSE.2020.2967380.

[5] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be ci skipped?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. DOI: 10.1109/TSE.2019.2897300.

[6] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, IEEE Press, 2019, pp. 105–115.

[7] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the International Conference on Mining Software Repositories (MSR 2020)*, 2020.

[8] C. Miller, D. Widder, C. Kästner, and B. Vasilescu, "Why do people give up FLOSSing? a study of contributor disengagement in open source," in *International Conference on Open Source Systems*, ser. OSS, Springer, 2019, pp. 116–129. DOI: https://doi.org/10.1007/978-3-030-20883-7_11.

[9] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, ser. MSR '17, 2017.

[10] D. Sundstrom, *Build #2954 - prestodb/presto - travis ci*, https://travis-ci.org/github/prestodb/presto/builds/41060068, (accessed on 02/24/2021), 2020.

[11] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 966–969.

[12] J. Nicolai, *Github welcomes all ci tools - the github blog*, https://github.blog/2017-11-07-github-welcomes-all-ci-tools/, (accessed on 01/26/2021), Nov. 2017.