A Large-Scale Empirical Study of Compiler Errors in Continuous Integration

Chen Zhang*
School of Computer Science
Fudan University
China

Bihuan Chen*†
School of Computer Science
Fudan University
China

Linlin Chen*
School of Computer Science
Fudan University
China

Xin Peng*
School of Computer Science
Fudan University
China

Wenyun Zhao*
School of Computer Science
Fudan University
China

ABSTRACT

Continuous Integration (CI) is a widely-used software development practice to reduce risks. CI builds often break, and a large amount of efforts are put into troubleshooting broken builds. Despite that compiler errors have been recognized as one of the most frequent types of build failures, little is known about the common types, fix efforts and fix patterns of compiler errors that occur in CI builds of open-source projects. To fill such a gap, we present a large-scale empirical study on 6,854,271 CI builds from 3,799 open-source Java projects hosted on GitHub. Using the build data, we measured the frequency of broken builds caused by compiler errors, investigated the ten most common compiler error types, and reported their fix time. We manually analyzed 325 broken builds to summarize fix patterns of the ten most common compiler error types. Our findings help to characterize and understand compiler errors during CI and provide practical implications to developers, tool builders and researchers.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Continuous Integration, Build Failures, Compiler Errors

ACM Reference Format:
Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A Large-Scale Empirical Study of Compiler Errors in Continuous Integration. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338917

*Also with the Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Institute of Intelligent Electronics & Systems, China.
†Bihuan Chen is the corresponding author.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338917

1 INTRODUCTION

Continuous integration (CI) is a software engineering practice of merging all the developers' working copies to a shared branch frequently [14]. The concept of CI was proposed in 1991 [10]. Then, it was adopted as one of the practices by Microsoft and Extreme Programming [5, 12, 17]. Gradually, CI gained wide acceptance due to its automated build process including compilation, static analysis and testing. CI can help developers detect and fix integration errors as early as possible, and reduce risks in software development [14].

With the widespread use and continued growth of CI [16, 24, 40], empirical studies have been recently conducted to explore the usage, cost, benefits, barriers and needs when developers use CI [23, 24, 49, 50]. They investigate the overall relationships between CI and software development, but are not designed to look into the details of CI builds. Specifically, CI builds often break (i.e., fail), and a large amount of efforts are put into troubleshooting broken builds [23, 29]. Therefore, studies have been conducted to analyze the type and frequency of build failures in industrial and open-source projects [36, 41, 53]. They find that test failures, violations in static analysis, and compiler errors are generally the most frequent types of CI build failures.

To have a deep understanding of one specific CI build failure type in open-source projects, Labuschagne et al. [31] focused on test failures, and Zampetti et al. [55] targeted violations in static analysis. However, little is known about the common types, fix efforts and fix patterns of compiler errors during CI in open-source projects. Such knowledge about compiler errors in CI is important for CI tools, developers and compilers. To the best of our knowledge, the only study on compiler errors is from Google [43]. However, it targets industrial projects but not open-source projects, and it does not systematically analyze the fix patterns of compiler errors.

To fill the gap in characterizing and understanding compiler errors during CI in open-source projects, we conducted a large-scale empirical study on 6,854,271 CI builds from 3,799 open-source Java projects hosted on GitHub and using Travis CI [4]. By analyzing the data, we answered four research questions in this paper.

- **RQ1: Frequency Analysis.** How often do CI builds fail because of compiler errors? (Sec. 4)
- **RQ2: Distribution Analysis.** What are the common types of compiler errors that cause CI build failures? (Sec. 5)
- **RQ3: Fix Effort Analysis.** How long does it take to fix CI build failures caused by compiler errors? (Sec. 6)

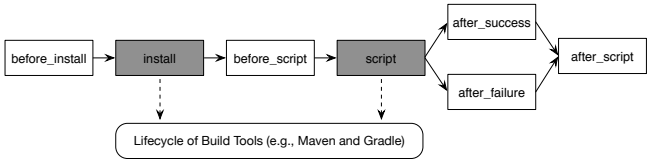


Figure 1: The Lifecycle of Travis CI

- **RQ4: Fix Pattern Analysis.** How are CI build failures caused by compiler errors fixed? (Sec. 7)

Our answers to the above four research questions can be summarized as the following findings. 11% of broken builds were a result of compiler errors, affecting 75% of projects. The ten most common compiler error types accounted for 90.2% of compiler errors. The median fix time of the ten most common error types ranged from 18 minutes to 97 minutes. Simple fix patterns did exist for most of the common error types, and IDEs provided limited supports to automatically fix them. Our findings shed light on the potential areas where developers, tool builders and researchers from the compiler and CI community can provide the most benefit.

In summary, this work makes the following contributions.

- We conducted a large-scale empirical study to understand the common types, fix efforts and fix patterns of compiler errors occurring during CI builds of open-source projects.
- We provided practical implications of our findings to three audiences: developers, tool builders and researchers.
- We released a large-scale dataset of broken CI builds to foster potential applications of this dataset.

2 CONTINUOUS INTEGRATION

Travis CI. An abundance of CI tools are available, from self-hosted to hosted systems [35]. For example, Jenkins [3] is a self-hosted system; i.e., developers need to set up the CI service locally, and Jenkins only store data on recent builds. Travis CI [4] is a hosted system, which is integrated with GitHub. The entire build history of a project is available via the Travis CI API. According to the analysis of 34,544 open-source projects from GitHub, 40% of projects use CI and 90.1% of them use Travis CI as their CI service [24]. To cover the majority of projects and have access to their entire build history, we focus our attention in this study on the projects that use Travis CI as their CI service.

Triggering Events. Travis CI builds can be triggered by several events. *Push* and *Pull Request* are the typical ones. When a commit is pushed to a repository on GitHub or a pull request is opened on GitHub, Travis CI triggers a build. Besides, Travis CI supports two other events: *API* and *Cron*. *API* means that developers can trigger a build by sending a post request directly to the Travis CI API. *Cron* means that builds are triggered at regular intervals (e.g., every week) independently of whether any commits were pushed to the repository.

Build Lifecycle. As shown in Fig. 1, the default lifecycle of Travis CI consists of multiple phases, among which install and script are the most important ones. The install phase installs any dependencies required for building a project. The script phase runs the build script [4] (e.g., running testing or static analysis tools). Since Travis CI relies on automated build tools such as Maven [1] and Gradle [2], the lifecyle of a CI build interacts with the lifecylce of a Maven or Gradle build, as shown by the dotted arrows in Fig. 1. Hereafter, a build refers to a CI build but not a Maven or Gradle build if not explained.

Maven and Gradle are designed to separate the compilation of production code and test code. By default, for the projects using Maven, both production and test code are compiled in the install phase; and thus compiler errors are reported in the install phase. Differently, for the projects using Gradle, the production code is compiled in the install phase, and the test code is compiled in the script phase; and thus compiler errors in the production code and test code are respectively reported in the install and script phase.

Jobs and States. In Travis CI, a build is a group of *jobs*, and a job is an automated build process that clones a repository into a virtual environment and carries out a series of phases such as compiling code, running static analysis, and executing tests. For example, a build has three jobs, each of which runs the build with a different JDK version. Hereafter, for the ease of presentation, a build refers to a CI build job.

Based on the result of a build, Travis CI assigns different *states* to a build. A build is marked as *passed* if the build ran successfully; *errored* if errors occurred in the before_install, install or before_script phase; and *failed* if errors occurred in the script phase. A build failure occurs if a build is errored or failed, and the build is called a broken build. A build might have other states (e.g., *canceled* and *started*).

3 EMPIRICAL STUDY METHODOLOGY

In this section, we first introduce the design of our empirical study, and then present our data collection process.

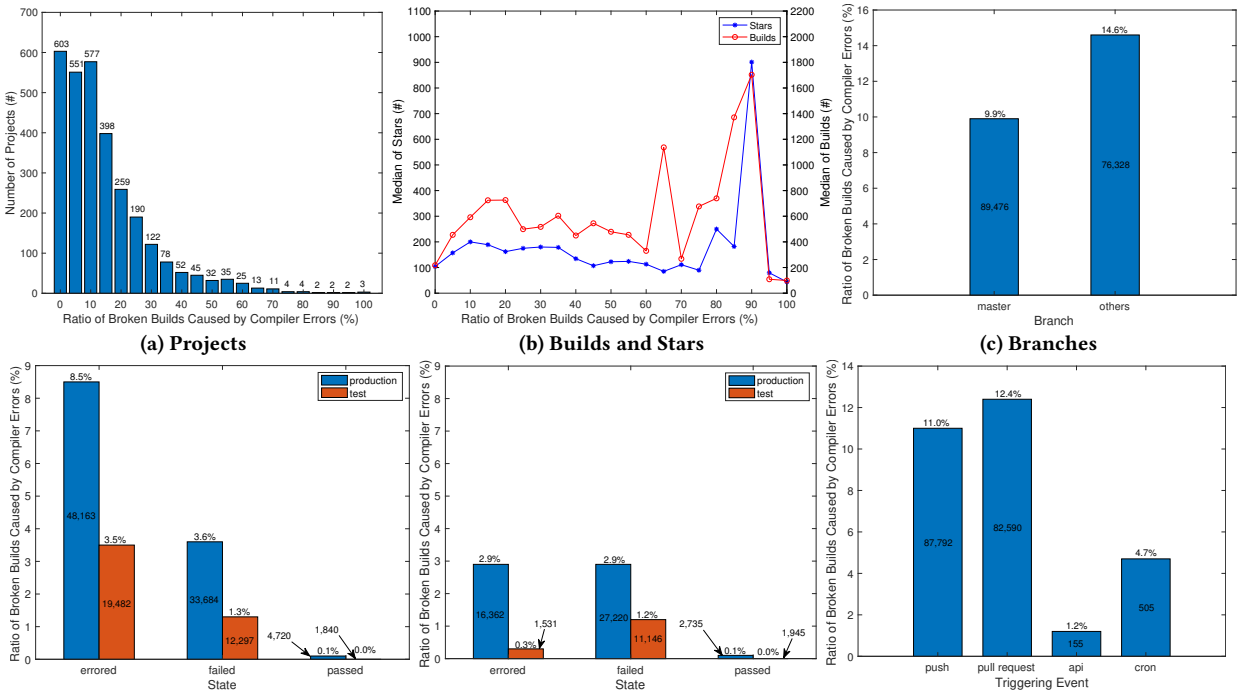
3.1 Study Design

Our goal is to characterize and understand the common types, fix efforts and fix patterns of compiler errors that break CI builds of open-source projects. To this end, we proposed the four research questions as introduced in Sec. 1.

The *frequency analysis* in RQ1 analyzes the overall frequency of broken builds caused by compiler errors, and the *distribution analysis* in RQ2 reports the overall distribution of compiler error types that break builds. Our findings from RQ1 can characterize the significance of compiler errors in broken builds and motivate the importance of our study. Our findings from RQ2 can identify the most common types of compiler errors where developers, researchers and tool builders from the compiler and CI community should concern.

Further, RQ1 and RQ2 investigate the frequency and distribution across three important aspects of open-source projects: project properties, development mechanisms, and CI mechanisms. For the three aspects, several factors are respectively included: number of projects, builds and stars, branches and code types, and build states and triggering events. Our findings can hint the circumstances where compiler errors are more likely to break builds and how compiler error types distribute differently, and suggest how to improve the development discipline and CI usage to avoid compiler errors during CI builds.

The *fix effort analysis* in RQ3 measures the time spent by developers fixing each type of compiler errors. Our findings from RQ3 can prioritize the types of compiler errors that may reduce developers' productivity and where automated debugging or fixing tools may provide the most benefit. The *fix pattern analysis* in RQ4 analyzes a total of 325 broken builds to summarize fix patterns of ten common types. Our findings from RQ4 can assess the feasibility of automatic fixing or fix recommendation of compiler errors and provide hints for developers to troubleshoot compiler errors during CI.



(a) Projects (b) Builds and Stars (c) Branches

(d) Code Types and Build States (Maven) (e) Code Types and Build States (Gradle) (f) Triggering Events

Figure 2: Frequency of Compiler Errors across Projects, Builds, Stars, Branches, Code Types, Build States and Triggering Events

3.2 Data Preparation

Since we focus on compiler errors occurring during CI builds of open-source projects, we first used the GitHub API to crawl the list of open-source Java projects that were not forked as of April 4, 2018. We focused on Java because it is widely-used and thus our findings can be beneficial to wide audiences. This resulted in a set of 1,703,090 projects. To ensure the quality of selected projects, we removed the projects that had less than 25 stars, which resulted in 23,693 projects. Of these projects, we selected the projects that used Travis CI and had more than 50 builds in order to provide sufficient build data, which restricted our selection to 3,872 projects. To obtain well formatted build logs for the ease of our analysis, we only kept the projects that used Maven or Gradle. Finally, we had a set of 3,799 projects.

We crawled the entire build history of the 3,799 project. The data of a build consists of two files: a *log* file with the logging information generated during the whole build process, which includes the information about compiler errors; and a *json* file with various fields (e.g., state) to report the statistics of a build. To this end, we crawled the data through a RESTful Web-API provided by Travis CI. Totally, we crawled a dataset of 6,854,271 builds for 3,799 projects, which had a total size of 3.8 TB. Then, we determined whether a build was broken (or successful) by checking whether the state field in its *json* file was *errored* or *failed* (or *passed*). Finally, we obtained 1,487,925 broken builds and 5,040,641 successful builds, which accounted for 95.2% of all builds. The other 4.8% had uncommon states (e.g., *canceled* and *started*) and were not considered due to their incomplete builds.

4 FREQUENCY ANALYSIS (RQ1)

To analyze the frequency of compiler errors, we obtained the broken builds that were caused by compiler errors. To this end, we applied a keyword search over *log* files of broken builds by checking whether the *log* file contains "COMPILATION ERROR" for Maven projects, or "Compilation failed" for Gradle projects. The two keywords are the default messages to indicate compiler errors in Maven and Gradle.

4.1 Overall Frequency

Of the 1,487,925 broken builds, we had 171,043 broken builds that were caused by compiler errors. They accounted for 11% of broken builds, which was above the 9% as reported by Vassallo et al. [53] (using 349 open-source Java projects) but below the 26% as reported by Miller et al. [36] (using 69 industrial build failures). The differences might owe to the different scale of the studies. Further, we analyzed the number of projects that had at least one broken build that was caused by compiler errors. As a result, 2,847 (75%) projects were affected by compiler errors. In that sense, compiler errors are a non-negligible failure types of broken builds, and affect most of the open-source Java projects, which motivates the study in this paper.

4.2 Frequency across Project Properties

We analyzed the ratio of broken builds caused by compiler errors for each project. As projects with only few broken build would introduce noise, we removed the 791 projects that had less than 20 broken builds. The result is reported in Fig. 2a, where the y-axis denotes the number of projects whose ratio is in a range (e.g., 10 represents the range (5, 10]). Of the 3,008 projects, 603 (20%) projects did not have any broken build caused by compiler errors; 1,785 (59%) projects had less than 20% of broken builds caused by compiler errors; and 620 (21%) projects had more than 20% of broken builds caused by compiler errors. Surprisingly, at least 50% of the broken builds in 101 (3%) projects were caused by compiler errors. We investigated these

projects and found that, they mostly do not have any tests or use any static analysis tools and hence broken builds are mostly caused by compiler errors. For the projects with no compiler errors, we found that some developers configure Travis CI incorrectly, making most builds break before the compilation, and they simply ignore the failures and do not fix the configuration. Hence, CI training and automated CI configuration tools are needed to ease the configuration and widen the practical adoption of CI in open-source projects.

In addition, we analyzed project's number of builds and stars to investigate whether they correlate to the ratio of broken builds caused by compiler errors. The result is reported in Fig. 2b, where the y-axis denotes the median number of builds and stars of the projects whose ratio is in a range. The projects whose ratio was between 0% and 20% had a larger median number of builds and stars (588 and 177) than the projects whose ratio was 0% (218 and 104) and than the projects whose ratio was between 20% and 100% (487.5 and 139). Thus, popular and frequently-built projects have a relatively low ratio of broken builds caused by compiler errors, and the differences are statistically significant (p = 2.8937e-5 and 3.0905e-5 in one-way ANOVA test [25]). It implies that frequent builds can facilitate fast integration. However, too frequent builds may incur high time overhead. Thus, build prediction techniques are needed to suggest developers when to build.

4.3 Frequency across Branches

We investigated the ratio of broken builds caused by compiler errors across different branches to test the hypothesis that master branches are less likely to contain compiler errors than non-master branches. We distinguished the master branch from others, and extracted this branch data from the branch field of the *json* files of broken builds. The result is shown in Fig. 2c, where the y-axis denotes the ratio and the number of broken builds caused by compiler errors is shown in each bar (which is the same in Fig. 2d, 2e and 2f).

We failed to get the branch data from the *json* files of 64,545 broken builds because of tag builds or missed build data; and 5,239 of them were caused by compiler errors. Of the 902,216 and 521,164 broken builds from the master and other branches, 9.9% and 14.6% were caused by compiler errors. This indicates that broken builds on master branches are less likely to have compiler errors than those on other branches. This difference is statistically significant (p = 2.3955e-14 in Wilcoxon signed rank test [45]). It is potentially because developers are often more careful when changing code on master branches so as to make master branches stable; otherwise, many developers will be affected. Hence, a strengthened discipline is need for committing to non-master branches, e.g., pre-compilation before committing.

4.4 Frequency across Code Types and Build States

As compiler errors in production and test code are reported in different phases of CI depending on whether Maven or Gradle is used (see details in Sec. 2), the build state of the resulting broken builds can be different. Hence, we analyzed the ratio of broken builds caused by compiler errors with respect to code types and build states together. We determined whether a broken build was caused by compiler errors in the production or test code by two ways: checking whether compiler errors occur in the default task of compiling production or test code as Maven and Gradle separate their compilation in two tasks, or checking whether the file with compiler errors locates in the default folder of production or test code as Maven and Gradle have separate folders for them. Further, we extracted the state data from the state field in the *json* files. Here we also analyzed the successful builds as we found some of them also had compiler errors. The results for projects using Maven and Gradle are shown in Fig. 2d and 2e, respectively.

We failed to decide whether compiler errors happened in the production or test code for 1,158 broken builds as some projects changed the default compilation tasks or source code structures. 564,563 and 923,362 broken builds were respectively marked as *errored* and *failed*, and 5,040,641 builds were marked as *passed*. First, production code was more likely to contain compiler errors than test code, and the difference is statistically significant (p = 2.3093e-235 in Wilcoxon signed rank test). The potential reasons are that, production code is usually more complex and frequently-changed than test code, and projects can have a small number of tests or even no test.

Second, for Maven projects, not all compiler errors occurred in *errored* builds; and for Gradle projects, not all compiler errors in production (resp. test) code occurred in *errored* (resp. *failed*) builds. This indicates that Travis CI provides the flexibility for developers to compile production code in the later script phase. However, this flexibility is at the cost of a longer build time, affecting the efficiency of software development. In detail, we analyzed the average build time of *errored* and *failed* builds that were caused by compiler errors; and *errored* builds took 212 seconds, while *failed* builds took 374 seconds. This difference is statistically significant (p = 0.0001 in one-way ANOVA test). Therefore, developers should be aware of this trade-off when changing compilation configurations.

Third, surprisingly, 11,239 of 5,040,641 successful builds also had compiler errors. To find root causes, two of the authors separately analyzed 96 cases by investigating commits, logs and configuration files to summarize root causes. Then, they discussed root causes and investigated inconsistent cases together to reach consensus. These 96 cases were randomly selected from 11,239 cases, achieving a confidence level of 95% and a margin of error of 10%. Finally, we identified three root causes: (i) compiler errors occur after the script phase, and they will not affect the build result (27%); (ii) Travis CI can be configured to allow some compiler errors to happen without affecting the build result (56%); and (iii) Travis CI provides a *retry* mechanism to rebuild when a previous build fails, and thus previously-occurred compiler errors would disappear in retried builds (17%). Thus, automated compiler error localization techniques are needed to locate such easily-missed but still risky compiler errors.

4.5 Frequency across Triggering Events

We investigated the ratio of broken builds caused by compiler errors across four kinds of events to trigger a build (see details in Sec. 2). We extracted this event data from the event_type field in the *json* files of broken builds. The result is presented in Fig. 2f.

It turned out that 31 broken builds had a value of *null* for this event data due to missed build data; and one of them was caused by compiler errors. Of the 796,827, 666,830, 13,439 and 10,798 broken builds respectively triggered by *Push*, *Pull Request*, *API* and *Cron*, 11.0%, 12.4%, 1.2% and 4.7% were caused by compiler errors. This shows that while being the most common way to trigger builds, pull requests are more likely to make compiler errors than pushes. The difference is statistically significant (p = 7.9891e-39 in Wilcoxon signed rank test). As pull

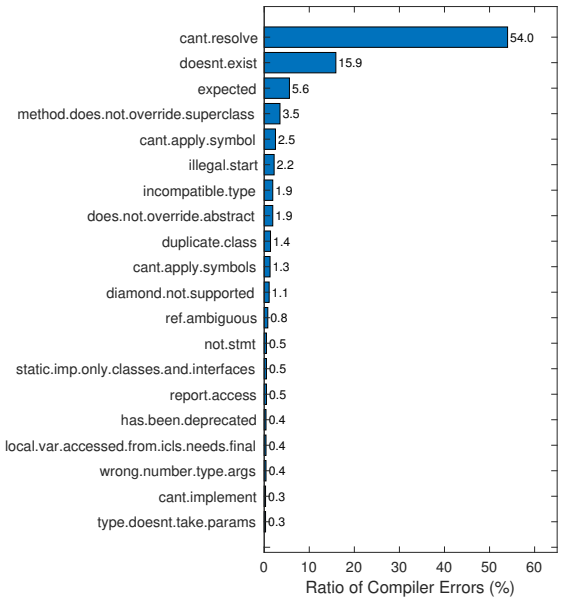


Figure 3: Overall Distribution of Compiler Error Types

requests are developed on an isolated branch from the master branch, developers may be less aware of code changes on the master branch. Besides, it is also possible to file a pull request for a feature that is incomplete such that other developers can provide suggestions inside of the pull request. On the other hand, as developers may do not use IDEs or be unaware of code changes from other developers, or different Java versions are used in local and CI compilation environments, compiler errors appear in pushes. Therefore, proactive alerting techniques are needed to inform developers about potential compiler errors just before pushes or pull requests.

> Compiler errors caused 11% of broken builds, affecting 75% of projects. 21% of projects had more than 20% of broken builds caused by compiler errors. Production code and pull requests were more likely to contain compiler errors than pushes and test code. Broken builds on master branches were less likely to be caused by compiler errors than those on other branches. Travis CI's flexibility to compile code in later phases was at a price of a longer build time. Even successful builds could contain compiler errors.

5 DISTRIBUTION ANALYSIS (RQ2)

To identify the common types of compiler errors and their distribution, we analyzed the compiler error messages reported in the *log* files of 171,043 broken builds caused by compiler errors. The list of compiler error types with the formatted error messages are available from the compiler properties file used by javac. Thus, we analyzed the compiler properties file for Java 6, 7, 8 and 9, made a union of their error types, and obtained 498 error types. Then for each of the error types, we wrote regular expressions based on the formatted error message and used them to match the error messages in the *log* files. Finally, we counted the number of matched compiler errors for each error type, and computed the distribution of error types.

5.1 Overall Distribution

We matched 252 error types with 1,785,067 compiler errors, but did not find any error for 246 types. We believe this is attributed to the extensive usage of IDEs during software development as many error types can be easily captured by IDEs and are fixed before building.

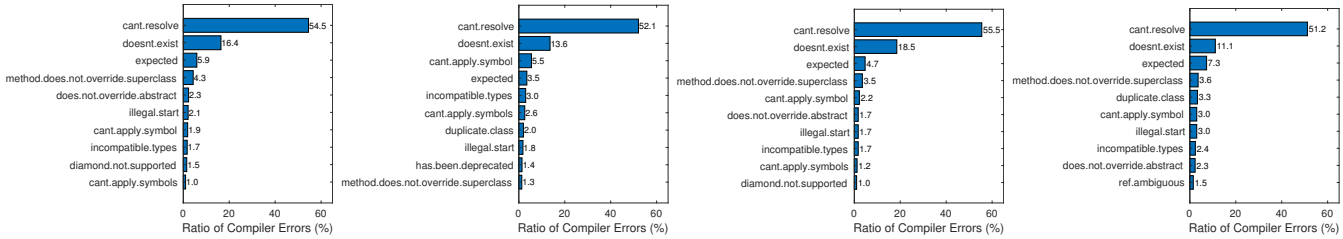
Fig. 3 lists the 20 most common error types. They covered 95.4% of compiler errors. The most common error type was cant.resolve that accounted for 54.0% of compiler errors. This error occurs when a compiler cannot recognize a symbol in code. doesnt.exist covered 15.9% of compiler errors. This error occurs when the compiler cannot find the imported package in the classpath. expected appeared in 5.6% of compiler errors. Such an error occurs when the compiler expects a token that is not found. The three most common error types covered 75.5% of compiler errors, while the ten most common ones covered 90.2%. Thus, developer, researchers and tool builders should focus on these most common error types to provide the most benefit. Hereafter, we will focus our discussion on the ten most common error types.

Comparison to Google's Study [43]. The most two common error types in ours are the same in theirs (i.e., they respectively rank 1st and 3rd as reported by Seo et al. [43]; and an error type strict ranks 2nd, which corresponds to a Google-specific dependency check). Such dependency related error types are common regardless of the different development practices in Google and open-source community. Besides, some common error types in ours are less common in theirs (e.g., expected and method.does.not.override.superclass rank 3rd and 4th in ours, but rank 9th and 8th as reported by Seo et al. [43]), or even not in their 20 most common error types (e.g., duplicate.class). We believe this attributes to different datasets produced via different development practices. Our dataset is produced from CI, while Google's dataset is produced from a monolithic software repository and a centralized build system where code review is conducted before committing and products are built from head. Thus, syntax errors (e.g., expected) and class access rule violation errors (e.g., method.does.not.override.superclass and duplicate.class) are less likely introduced by Google's developers. Further, some common error types in theirs are missing in our 20 most common error types (e.g., unchecked and rawtypes rank 7th and 13th as reported by Seo et al. [43], but rank 39th and 30th in ours). As unchecked and rawtypes are warnings, only 72 projects in our study were configured to treat them as errors and the other projects just filtered them out, but we believe the command line flags to `javac` in the centralized build system in [43] were consistent across the dataset. These differences motivate the necessity of this study.

5.2 Distributions across Code Types and Triggering Events

We further analyzed whether the distribution of compiler error types differs with respect to projects, branches, code types, build states and triggering events. We found that the distribution had no statistically significant difference for projects whose ratio of broken builds caused by compiler errors was below and above 20% (see Sec. 4.2) and for different build states and branches. Hence, we omitted their results; and investigated the ten most common compiler error types for different code types and triggering events.

Fig. 4a and 4b report the results for production code and test code. The first major difference was that does.not.override.abstract and



(a) Production Code **(b) Test Code** **(c) Pushes** **(d) Pull Requests**

Figure 4: Distribution of Compiler Error Types across Code Types and Triggering Events

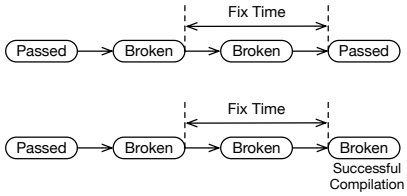


Figure 5: Examples of Computing Fix Time

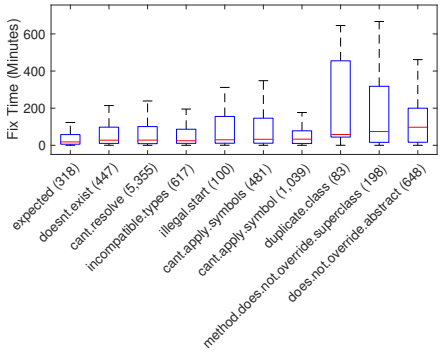


Figure 6: Fix Time of Common Compiler Error Types

method.does.not.override.superclass were more commonly seen in production code than in test code. This difference is statistically significant (p = 7.0709e-26 and 9.4066e-47 in Wilcoxon signed rank test). These two error types are related to *inheritance* and *implements*. Generally, production code has more complicated *inheritance* and *implements* relationships than test code, which might cause the difference. The second difference was that cant.apply.symbol occurred more frequently in test code than in production code. This error occurs when a method is called with an incorrect argument list. This difference is statistically significant (p = 2.1176e-14 in Wilcoxon signed rank test). This can be attributed to that test code may not be timely maintained when production code changes. Hence, automated co-evolution techniques are needed to either suggest or automate the co-evolution of production code and test code.

Fig. 4c and 4d present the results for pushes and pull requests. One main difference was that doesnt.exist was less commonly seen in pull requests than in pushes. This difference is statistically significant (p = 1.6670e-15 in Wilcoxon signed rank test). As pull requests are filed with the expectation to be successfully merged, developers are less likely to import a non-existent package. Besides, pushes have a high chance to add new features that often introduce new classes.

> cant.resolve, doesnt.exist, and expected were the most common compiler error types. The ten most common compiler error types accounted for 90.2% of compiler errors.

6 FIX EFFORT ANALYSIS (RQ3)

To measure the fix effort of each compiler error type, we analyzed the elapsed time during which a broken build (caused by compiler errors) was fixed to be either *passed* or *failed* with its compilation being successful (i.e., the previous compiler errors had been fixed, but the build still broke, e.g., due to test failures), as shown in Fig. 5. The build can break for several times before compiler errors are fixed. Therefore, we identified the *sequences of broken builds* that had one type of compiler errors and were succeeded by a successful build or a failed build with a successful compilation. Since the sequences of Travis CI builds are not linear but a directed graph [8], we followed the method proposed by Beller et al. [8] to extract the sequences. Following Google's study [43], we excluded broken builds that had multiple types of compiler errors because it is difficult to discriminate the fix time for each error type. In total, we found 17,948 sequences. Then, we computed the fix time as the time interval between the finishing time of the first broken build in the sequence (i.e., indicating the discovery of a compiler error) and the creation time of the succeeded build in the sequence. As suggested in Google's study [43], we excluded the sequences in which the fix time was greater than 12 hours to avoid compounding the fix time with developers' schedule as well as to have fair comparisons with Google. Finally, we computed the fix time for 12,000 sequences, covering 138 error types.

Fig. 6 presents the fix time of the ten most common compiler error types in Fig. 3. A box plot is shown for each error type, and the line within the box denotes the median value of fix time. The error types in Fig. 6 were ordered in increasing median fix time, and the number of identified sequences that contained the error type was reported in the parentheses after each error type name in the x-axis.

Overall, the median fix time of the error types ranged from 18 minutes to 97 minutes. Some error types like does.not.override.abstract and method.does.not.override.superclass took twice more fix time than others such as expected, doesnt.exist and cant.resolve. This difference is statistically significant (p = 3.1464e-19 in one-way ANOVA test). This is reasonable because both does.not.override.abstract and method.does.not.override.superclass usually require developers to think about the implementation of new methods, while expected involves token-level syntax changes and doesnt.exist and cant.resolve are mostly related to dependency or type mismatch issues. Besides, the median number of build attempts until the compiler errors were fixed was 1; and 25.9% of the compiler errors were fixed by at least 2 builds. This indicates that compiler errors are not trivial to fix.

Comparison to Google's Study [43]. The median fix time of the common compiler error types in our study was much longer than in

Google's study (less than 12 minutes). This large difference can be attributed to that Google's dataset is gleaned from iterative development and Google's monolithic repository asks developers to quickly respond to errors to avoid affecting a wide range of developers.

> The median fix time of the ten most common compiler error types ranged from 18 minutes to 97 minutes.

7 FIX PATTERN ANALYSIS (RQ4)

To analyze the fix patterns of compiler errors, we focused on the ten most common error types. For each error type, we first randomly selected five broken builds that contained the error type, and two of the authors separately looked into the broken builds (with the code, commit or pull request) to investigate how they were fixed. Then, they discussed their discovered fix patterns to reach a consensus. Multiple rounds of selection, investigation and discussion were conducted until no new fix pattern could be discovered for two consecutive rounds. Finally, we analyzed a total of 325 broken builds. Table 1 lists the derived fix patterns for each error type, where the first column shows the error type with the number of analyzed broken builds in the parentheses, the second column describes the error, the third column shows the fix pattern with the number of fixed broken builds in the parentheses, and the last column reports the number of errors falsely reported or having imprecise error message.

cant.resolve. We analyzed 90 broken builds to have stable fix patterns for cant.resolve. The most common fix is to correct mistypes, e.g., change package, class, method or variable name. The error is usually due to misspelling, incomplete refactoring or code change, usage of snapshot libraries, and different Java versions in build environments. Another common fix is to remove all the relevant code about the symbol not found, which is often caused by incomplete refactoring or code change. An interesting pattern is to add a class, meaning that developers seem to use a class that does not exist. A close look at the commit messages shows that developers often miss some code files in commits. Adding an import statement or a dependency library is also quite common. Less common patterns include adding a variable declaration, updating the dependency library version, moving a class from a package to another or casting the object type.

doesnt.exist. From 30 broken builds, we summarized four patterns to fix doesnt.exist. In particular, adding a dependency library is the most common pattern, which resolves errors caused by a missed dependency declaration in automated building tools. Changing an imported package fixed eight broken builds. These errors are often due to misspelling, incomplete code change, or usage of snapshot libraries. Similar to the case in cant.resolve, developers may miss to commit a whole package. Thus, a corresponding fix pattern is to add the package and the classes in it. The last pattern is to remove the package import statement. It is interesting that in one of the broken builds, a cant.resolve error was falsely reported as a doesnt.exist error due to static method invocation, where the compiler falsely considered a class as a package since it is possible that a class is used through its fully qualified name without an import statement.

expected. We investigated 35 broken builds of expected. As this error type is generally related to syntax violations, many fixes are hard to summarize, e.g., changing a token from "," to ";", add "{", or remove ")". Therefore, we grouped them together as a pattern to change, add or remove a token. Besides, a common pattern is to remove illegal tokens, which occurred 16 times. Most of the illegal tokens were resulted from merge conflicts, which developers did not resolve. The other two patterns are to change misspelled keywords and move code (e.g., moving a method declaration out of a method declaration). Surprisingly, 29 (82.9%) broken builds had imprecise error messages that failed to locate the position of an error.

method.does.not.override.superclass. Via analyzing 25 broken builds, we summarized seven fix patterns. Most commonly, it is fixed by removing the @Override annotation and method in the subclass, which is usually caused by the removed method in its superclass (i.e., incomplete code changes). Another common fix is to change the method signature in the subclass to match the already changed method in its superclass. Adding the superclass fixed two broken builds, where the cant.resolve error was reported together. Updating dependency library version fixed two broken builds. The libraries were developed by developers in the same team, but violated the original design. Thus, the libraries were updated to follow the original design. Other patterns are to remove the @Override annotation, or add the missed superclass or dependency library.

cant.apply.symbol(s). For the 40 broken builds, the common fix patterns are all about changing arguments in a method call, i.e., add an argument, remove an argument, change the type of an argument, and change the order of two arguments. These errors are caused by incomplete refactoring or code changes, or method misuses due to carelessness or unfamiliarity. Another common fix is to add missed method declaration, which is usually caused by new features. A common pattern is to change the method name, which means that developers called the wrong method. A less common pattern is to remove the method call. We got one false reports due to a bug in type inference in Java generics, and it was resolved by updating Java version.

illegal.start. Similar to expected, illegal.start is related to syntax violations, and often occurs simultaneously with expected. Its fix patterns are almost the same to expected except that it has a pattern that adds the generic type in constructor or method invocations. Actually, generic types can be omitted in Java 7 and beyond. This error occurs because of a lower Java version in CI build environment.

incompatible.types. This error type usually occurs in assignments, where the right hand side can be a literal, a variable, a method invocation, etc. Among 25 broken builds, the most common fix pattern is to change the assignee's declared type to match the previously-changed type of the right hand side. Correspondingly, some fix patterns are to change the type of right hand side, e.g., change method's return type and its implementation, assign a new value, and convert the type of a method call's return value through a chained call (e.g., a chained call to toString()). Other fix pattern are to add a generic type when the generic type inference fails to work, or to simply remove the assignment code resulted from incomplete code changes. Similar to cant.apply.symbol(s), we got two false reports due to the same bug in type inference in Java generics.

does.not.override.abstract. Analyzing 25 broken builds, we derived three fix patterns for does.not.override.abstract. The most common one is to implement the abstract method in the subclass. Such errors are mostly caused by the new features added in the superclass. Because of incomplete refactoring or code changes, the method signature in the superclass is changed without timely changing the

Table 1: Fix Patterns of the Common Compiler Error Types

Error Type	Error Description	Fix Pattern	Eclipse		IntelliJ IDEA		Imprecise
			Sug.?	Aut.?	Sug.?	Aut.?	
cant.resolve (90)	symbol (e.g., class, variable, or method) is not found	correct mistype (24)	✓	×	✓	×	0
		remove relevant code (22)	✓	×	✓	×	
		add class (14)	✓	×	✓	×	
		add import (9)	✓	✓	✓	✓	
		add dependency library (8)	✓	×	✓	×	
		add method (6)	✓	×	✓	×	
		add variable (4)	✓	×	✓	×	
		update dependency library (1)	×	×	×	×	
		move class (1)	×	×	×	×	
		cast object type (1)	✓	✓	✓	×	
doesnt.exist (30)	package does not exist	add dependency library (10)	✓	×	✓	×	1
		change imported package (8)	✓	×	✓	×	
		add imported package and class (8)	✓	×	✓	×	
		remove package import (4)	✓	×	✓	×	
expected (35)	token is expected but not found	remove illegal tokens (16)	×	×	×	×	29
		change, add or remove token (15)	×	×	×	×	
		change keyword (2)	×	×	×	×	
		move code (2)	×	×	×	×	
method.does.not.override.superclass (25)	method has @Override annotation but does not override or implement method from superclass	remove @Override and method (10)	×	×	×	×	0
		change method signature in subclass (8)	✓	×	✓	×	
		add superclass (2)	✓	×	✓	×	
		update dependency library (2)	×	×	×	×	
		remove @Override (1)	✓	×	×	×	
		import superclass (1)	✓	✓	✓	✓	
		add dependency library (1)	✓	×	✓	×	
cant.apply.symbol(s) (40)	method is called with incorrect argument list	add argument (13)	✓	×	×	×	1
		remove argument (8)	✓	×	✓	×	
		change argument type (6)	✓	×	✓	×	
		add method (5)	✓	×	✓	×	
		change method name (4)	✓	×	×	×	
		change argument order (2)	✓	✓	✓	×	
		remove method call (1)	×	×	×	×	
illegal.start (30)	start of expression, type or statement is illegal	remove illegal code (17)	×	×	×	×	0
		change, add or remove token (9)	×	×	×	×	
		add generic type (2)	✓	✓	✓	✓	
		move code (2)	×	×	×	×	
incompatible.types (25)	types are incompatible in assignment	change assignee's type (5)	✓	×	✓	×	2
		assign new value (4)	×	×	×	×	
		remove relevant code (4)	×	×	×	×	
		convert return value's type (4)	✓	×	✓	×	
		change method's return type (3)	✓	✓	✓	✓	
		add generic type (3)	✓	×	✓	×	
does.not.override.abstract (25)	method is not abstract and does not override abstract method from superclass	implement method in subclass (18)	✓	×	✓	×	0
		change method signature in subclass (5)	✓	×	✓	×	
		remove method in superclass (2)	×	×	×	×	
duplicate.class (25)	two classes have the same fully qualified name	remove class (14)	×	×	×	×	0
		change package name (7)	✓	×	✓	×	
		change class name (4)	✓	✓	✓	×	

method signature in the subclass. Therefore, the corresponding fix is to accordingly change the method signature in the subclass. Another common pattern is to remove the method in the superclass, which occurs due to the redesign of interfaces.

duplicate.class. We analyzed 25 broken builds of duplicate.class. The most common fix pattern is to remove the duplicated class. Such errors are usually caused by branch merges or a lack of coordination among developers. Other patterns are to change the class name or package name to resolve naming conflicts among developers.

From this manual analysis, we found that while fix patterns often exist for most of the common error types, the root causes of the errors are often hard to derive, simply from build logs or commits, due to the lack of context when developers make such errors. Moreover, most of the fix patterns were simple, only involving lines of code changes or code removals. Such simple fix patterns indicate the potential feasibility of automatically fixing common compiler errors. For example, "add dependency library" can be implemented by locating the jar file containing the specific class from the maven repository. However, the main challenge is to automatically identify the root cause such that we can know which fix pattern should be applied. On the other hand, such fix patterns and the potential root causes can be useful for developers to troubleshoot compiler errors or avoid some compiler errors during collaborative development.

Modern IDEs already provide the capability of fix suggestion and automated fix to compiler errors. Therefore, we reproduced the compiler errors in 325 broken builds in the two most widely adopted IDEs [44], Eclipse and IntelliJ IDEA, to (i) determine whether they provide our derived fix patterns as fix suggestions and (ii) apply suggested fixes to see whether they automatically fix compiler errors without any human intervention. The results are reported in the fourth to seventh columns of Table 1, respectively under column *Sug.?* and *Aut.?*. Of the 48 fix patterns, Eclipse and IntelliJ IDEA respectively do not support 16 and 19 fix patterns. Specifically, they have poor supports for syntax violation errors (e.g., expected and illegal.start). Among the supported 32 and 29 patterns, Eclipse and IntelliJ IDEA only support automated fix for 7 and 4 fix patterns. Therefore, our derived

patterns can be used to enrich fix suggestions in IDEs, and more importantly, automated techniques are needed to automatically fix compiler errors so as to improve IDEs.

> Simple fix patterns did exist for most of the ten common compiler error types, but root causes were hard to analyze due to the lack of development context. IDEs provided very limited supports to automatically fix compiler errors.

8 DISCUSSION

In this section, we discuss the threats to the validity of our empirical study and the practical implications of our findings.

8.1 Threats to Validity

Construct. We are interested in compiler errors that break CI builds of open-source projects. Hence, we designed four research questions to analyze the frequency, common types, fix efforts and fix patterns of compiler errors. We believe these questions have the potential to provide implications to different audiences (see Sec. 8.2). Besides, we broke down the results of frequency and common types across three aspects that capture the properties of projects and how the projects are developed and use CI. We selected several factors for each aspect. While not to be exhaustive, the factors capture key characteristics of each aspect. Our purpose is to identify the opportunities to improve development discipline and CI usage to reduce compiler errors.

Internal. This study is mostly focused on the ten most common compiler error types. They accounted for more than 90% of compiler errors. Such a high coverage indicates the representation of our results. More specifically, in RQ2, we focused on the error types in compiler properties files of Java 6, 7, 8 and 9. While we cannot guarantee that all 3,799 projects do not use lower Java versions, we believe this choice is representative as the public updates of Java 5 ended in November 2009 and most error types across Java versions are the same.

In RQ3, the computed fix time for each error type might not precisely reflect the time developers spent in fixing errors as developers often tangle multiple development tasks and switch between tasks. To mitigate this threat, we excluded build sequences whose fix time was greater than 12 hours. Besides, the identified sample data for each error type varied by an order of magnitude, e.g., ranging from 83 to 5,355. This is due to the unbalance distribution across error types and our constraints on identifying the sample data for a relatively precise computation of fix time. We believe this trade-off is reasonable and at least 83 sample data can provide representative results.

In RQ4, we analyzed 325 broken builds to manually summarize fix patterns for each common error type. We cannot guarantee that the derived fix patterns are complete, but we believe they can have a good coverage because we performed several rounds of data selection, investigation and discussion until no new fix pattern was discovered.

External. We analyzed a large number of 3,799 Java projects with a total of 6,854,271 CI builds. While believing that our findings can be generalized for Java projects, we cannot guarantee that our results can still hold for projects in different programming languages (e.g., C++ and C#). This is due to the specific compilers designed for different languages. Further studies are needed to investigate such differences. Besides, we only used Java projects that adopted Travis CI. As Travis CI is widely-used in more than 90% of GitHub projects using CI [24], we believe our findings are representative.

8.2 Implications

Developers. This study identifies the common compiler error types where developers should be pay attention to and need the most help. Moreover, our findings provide some development suggestions. Developers should know the trade-off when changing compilation configurations; developers should analyze whether test code needs co-evolution when changing production code; developers should guarantee the integrity of committed changes to avoid missing some files; developers should resolve merge conflicts timely; developers should strength the discipline of team coordination to avoid dependency issues (e.g., some developers included dependencies in their local classpath, but did not put them in configuration files, which might cause errors in CI builds due to the missing dependencies); and developers should be aware of the differences between local development environments and CI build environments (e.g., the differences in JDK versions, and the accessibility of dependencies). Our fix patterns can be useful for developers to troubleshoot compiler errors.

Tool Builders. Our findings in studying common compiler error types and their fix time shed light on the areas where tool builders can provide the most benefit for developers. For CI community, we found projects that incorrectly configured CI but never fixed the configuration problem, and projects that configured CI to perform compilation at later phases at a cost of a longer build time. This shows that Travis CI's configuration flexibility may have negative effects. Hence, training or automated configuration tools are needed to lower the entry barrier of CI or take the full advantage of CI. Besides, tools are needed to automatically predict and suggest when to build projects to ensure fast integration and low time overhead, and to parse build logs to automatically locate compiler errors in failed or even passed builds. For compiler community, we found that some compiler errors were falsely reported, and some error messages failed to report precise error positions or hints. Based on such cases, designers can find hints to build more powerful compilers. Besides, IDE developers can use our fix patterns to enrich their fix suggestions.

Researchers. This study highlights the needs for researchers to propose new techniques to automatically debug, locate or fix common compiler errors, which can be helpful to developers. Considering the recent advances in automatic program repairing [33, 39, 54], we believe there is a high potential of fixing compiler errors. Along this line, Santos et al. [42] have attempted to fix single token syntax errors (i.e., some of the errors in expected and illegal.start). Our summarized fix patterns can provide them with hints on automatic fixing, while our dataset of compiler errors can serve as a benchmark to evaluate fixing techniques. Moreover, a systematic study of the root causes of compiler errors can be helpful. Experiences from our manual analysis suggest that interviews with developers or analyzing developers' programming activities might provide good insights. Besides, pro-active alerting techniques should be proposed to inform developers about potential compiler errors before committing.

9 RELATED WORK

CI Studies. Vasilescu et al. [49] analyzed the popularity of CI using 233 GitHub projects, and found that pull requests are much more

likely to result in successful builds than pushes. Vasilescu et al. [50] also studied the productivity and quality for 246 GitHub projects using CI, and found that CI helps to merge pull requests from core members more effectively and detect more bugs by core developers.

Hilton et al. [24] conducted a large-scale quantitative study of the usage, costs and benefits of CI in open-source projects. They found that CI is widely used by the most popular projects; the lack of familiarity is the most common reason of not using CI; and CI helps projects to release twice as often and accept pull requests faster. Hilton et al. [23] further studied the barriers and needs when adopting CI. They derived three trade-offs developers faced, i.e., testing speed and certainty, access to the CI system and information security, and more configuration options and greater ease of use.

Gautam et al. [20] adopted clustering analysis to classify projects that follow CI practices and have distinct characteristics with respect to activity, popularity, size, testing and stability, using the TravisTorrent dataset [8] from 1,300 projects. Besides, industrial case studies (e.g., [32, 36, 46, 47]) were reported with respect to the adoption of CI and corresponding lessons learned. Recently, Vassallo et al. [51] analyzed whether and how the adoption of CI has changed the way developers perceive and adopt refactoring; and Bernardo et al. [9] studied how CI has affected the time to deliver merged pull requests.

These studies discuss the overall relationships between CI and software development, but are not designed to look into the details of CI builds. Instead, our study is focused on one of the most frequent types of build failures, i.e., compiler errors.

CI Build Failures. Miller's seminal work [36] on CI build failures in Microsoft products categorized 69 build failures into compilation (26%), unit tests (28%), static analysis (40%) and server failures (6%). Rausch et al. [41] studied CI build failures in 14 GitHub Java projects and found 14 failure types. They reported that testing, code quality and compilation are the most frequent types. Vassallo et al. [53] compared CI build failures in 349 Java open-source projects and 418 projects in an industrial organization. Their results showed that testing, compilation and dependency are the most frequent failure types for open-source projects; and testing, release preparation and static analysis are the most frequent failure types for industrial projects. These studies inspired us to specifically focus on compiler errors as they showed the significance of compiler errors in CI.

Instead of classifying failure types, Kerzazi et al. [29] investigated the impacts, circumstances and factors of build failures by analyzing 3,214 builds in a large software company in a period of six months. They showed that build failures cost more than 336.18 man-hours, are mostly caused by missing files, accidental commits and missing transitive dependencies. Gallaba et al. [18] analyzed the noise in build data. They found that 12% of passing builds had an actively ignored failure, while 9% of builds had a misleading or incorrect outcome. Instead of analyzing all build failure types together, we focus on one specific failure type in this study.

Hassan et al. [21] studied the feasibility of automatic building using 200 GitHub projects and analyzed the root causes of build failures. Another similar work was studied by Tufano et al. [48]. However, they both targeted traditional build environments, but not CI. Thus, they run default build commands of build tools to generate build logs, but we collect real-life build logs from Travis CI servers.

Compilation in CI. Our work falls in the compilation step of CI, and the closest study is by Seo et al. [43]. They investigated failure frequency, error types and fixing efforts of compilation, using 26.6 million builds produced in nine months by thousands of Java and C++ developers in Google. Different from industrial projects, open-source Java projects are used in our study. Moreover, we analyze the frequency and error types across three aspects and summarize the fixing patterns of common compiler errors, but they only analyzed fix patterns for 25 cant.resolve errors.

Static Analysis in CI. Zampetti et al. [55] studied how static analysis tools are used in CI pipelines using 20 GitHub projects. In these projects, seven static analysis tools are used; build failures caused by static analysis are mainly related to adherence to coding standards; and build failures related to detected potential bugs or vulnerabilities occur less frequently. Usage of static analysis tools in open-source projects has been widely studied (e.g., [6, 28, 30, 56]), whose results may not hold in CI. These studies are focused on the static analysis step in CI, but we focus on the compilation step.

Testing in CI. Beller et al. [7] studied the testing adoption in CI, using 1,359 GitHub projects. They reported that only around 20% of projects never include testing in CI; failing tests are the dominant reason for build failures; programming language has a strong impact on the number of tests and test failures; and multiple integration environments help to uncover more test failures. Labuschagne et al. [31] investigated the cost and benefits of regression testing in CI, using 61 GitHub projects. Among the 87% non-flaky test failures, 74% are caused by bugs, which can be seen as the benefits; and 26% are due to incorrect or obsolete tests, which represent the maintenance cost. Moreover, several advances have been made on studying ways to improve testing in CI [11, 13, 15, 27, 37, 38, 57]. These studies target the testing step in CI, and we study the compilation step.

Failure Build Repair. Recently, researcher have studied automatic methods to repair build failures. Hassan and Wang [22] proposed to repair build scripts using fix patterns extracted from existing build script fixes using predefined fix-pattern templates. Vassallo et al. [52] provided repair hints by summarizing failed build logs and linking StackOverflow discussions. Macho et al. [34] targeted dependency-related build failures and summarized three strategies (i.e., update version, delete dependency and add repository) to repair them. Santos et al. [42] applied n-gram and LSTM models to fix *single token* syntax errors, which is useful especially for novice programmers [26]. Gallaba et al. [19] studied how CI features in configuration files are used and misused, and derived four anti-patterns. They also developed tools to detect and remove these anti-patterns in CI configuration files. We believe our study can shed light on automatic repairing of compiler errors.

10 CONCLUSIONS

In this paper, we present a large-scale empirical study to characterize and understand the compiler errors that break CI builds of open-source projects. Our findings provided practical for developers, tool builders and researchers. In future, we plan to investigate how to automatically fix compiler errors. Our dataset with the more complete analysis results are released at https://compilererrorinci.github.io.

ACKNOWLEDGMENT

This work was supported in part by the National Key Research and Development Program of China (2016YFB1000801).

REFERENCES

[1] [n.d.]. Introduction to the Build Lifecycle of Maven. Retrieved February 15, 2019 from https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

[2] [n.d.]. The Java Plugin of Gradle. Retrieved February 15, 2019 from https://docs.gradle.org/current/userguide/java_plugin.html

[3] [n.d.]. Jenkins User Documentation. Retrieved February 15, 2019 from https://jenkins.io/doc/

[4] [n.d.]. Travis CI Docs. Retrieved February 15, 2019 from https://docs.travis-ci.com/user/getting-started/

[5] Kent Beck. 1999. Embracing change with extreme programming. *Computer* 32, 10 (1999), 70–77.

[6] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 470–481.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 356–367.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 447–450.

[9] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. 2018. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories*. 131–141.

[10] Grady Booch. 1991. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc.

[11] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 55–66.

[12] Michael A. Cusumano and Richard W. Selby. 1995. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press.

[13] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. 2012. Communicating continuous integration servers for increasing effectiveness of automated testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 374–377.

[14] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[15] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.

[16] Nicole Forsgren, Gene Kim, Jez Humble, Alanna Brown, and Nigel Kersten. 2017. 2017 State of DevOps Report. https://www.ipexpoeurope.com/content/download/10069/143970/file/2017-state-of-devops-report.pdf

[17] Martin Fowler. 2000. Continuous Integration. http://martinfowler.com/articles/originalContinuousIntegration.html

[18] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 87–97.

[19] Keheliya Gallaba and Shane McIntosh. 2018. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* (2018).

[20] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 495–498.

[21] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 38–47.

[22] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering*. 1078–1089.

[23] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 197–207.

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

[25] David C. Howell. 2012. *Statistical Methods for Psychology* (8 ed.). Cengage Learning.

[26] Matthew C. Jadud. 2006. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*. 73–84.

[27] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*. 712–723.

[28] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. 672–681.

[29] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. 41–50.

[30] Sunghun Kim and Michael D Ernst. 2007. Which warnings should i fix first?. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 45–54.

[31] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 821–830.

[32] Eero Laukkanen, Maria Paasivaara, and Teemu Arvonen. 2015. Stakeholder Perceptions of the Adoption of Continuous Integration–A Case Study. In *Proceedings of the Agile Conference*. 11–20.

[33] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*. 3–13.

[34] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 106–117.

[35] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE software* 31, 3 (2014), 14–16.

[36] Ade Miller. 2008. A Hundred Days of Continuous Integration. In *Proceedings of the Agile Conference*. 289–293.

[37] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2015. Preventing Data Errors with Continuous Testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 373–384.

[38] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2013. Data debugging with continuous testing. In *Proceedings of the 2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 631–634.

[39] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. 772–781.

[40] V One. 2017. 11th Annual State of Agile Development Survey. http://www.agile247.pl/wp-content/uploads/2017/04/versionone-11th-annual-state-of-agile-report.pdf

[41] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 345–355.

[42] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 311–322.

[43] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. 724–734.

[44] Oleg Shelajev. 2017. RebelLabs Developer Productivity Report 2017: Why do you use the Java tools you use? https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/

[45] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.

[46] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48–59.

[47] Sean Stolberg. 2009. Enabling agile testing through continuous integration. In *Proceedings of the Agile Conference*. 369–374.

[48] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017), 1–11.

[49] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. 2014. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. 401–405.

[50] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 805–816.

[51] Carmine Vassallo, Fabio Palomba, and Harald C Gall. 2018. Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 564–568.

[52] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break My Build: Assisting Developers with Build Repair Hints. In *Proceedings of the 26th International Conference on Program Comprehension*. 41–51.

[53] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*. 183–193.

[54] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*. 416–426.

[55] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 334–344.

[56] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32, 4 (2006), 240–253.

[57] Celal Ziftci and Jim Reardon. 2017. Who broke the build?: automatically identifying changes that induce test failures in continuous integration at Google scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. 113–122.