Evaluating Test-Suite Reduction in Real Software Evolution

August Shi
University of Illinois
Urbana, IL, USA
awshi2@illinois.edu

Alex Gyori
University of Illinois
Urbana, IL, USA
gyori@illinois.edu

Suleman Mahmood
University of Illinois
Urbana, IL, USA
msm6@illinois.edu

Peiyuan Zhao
University of Illinois
Urbana, IL, USA
pzhao12@illinois.edu

Darko Marinov
University of Illinois
Urbana, IL, USA
marinov@illinois.edu

ABSTRACT

Test-suite reduction (TSR) speeds up regression testing by removing redundant tests from the test suite, thus running fewer tests in the future builds. To decide whether to use TSR or not, a developer needs some way to *predict* how well the reduced test suite will detect real faults in the future compared to the original test suite. Prior research evaluated the cost of TSR using only program versions with seeded faults, but such evaluations do not explicitly predict the effectiveness of the reduced test suite in future builds.

We perform the first extensive study of TSR using real test failures in (failed) builds that occurred for real code changes. We analyze 1478 failed builds from 32 GitHub projects that run their tests on Travis. Each failed build can have multiple faults, so we propose a family of mappings from test failures to faults. We use these mappings to compute *Failed-Build Detection Loss* (FBDL), the percentage of failed builds where the reduced test suite misses to detect all the faults detected by the original test suite. We find that FBDL can be up to 52.2%, which is higher than suggested by traditional TSR metrics. Moreover, traditional TSR metrics are not good predictors of FBDL, making it difficult for developers to decide whether to use reduced test suites.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software evolution**;

KEYWORDS

Test-suite reduction, regression testing, continuous integration

ACM Reference Format:
August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213875

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213875

1 INTRODUCTION

Regression testing is the act of retesting modified software. Developers use regression testing to quickly detect code changes that introduce bugs in their code. Regression testing is commonly done on continuous integration (CI) servers in the cloud. For example, when a developer pushes some code changes to GitHub, Travis can run a *build* in the cloud. If the build fails due to test failures, the developer inspects the failures to find the faults that need to be fixed. Regression testing is beneficial and widely practiced, but it can be costly, because many tests run on every code change, and changes occur very frequently [48]. In addition to time, there is a monetary cost in utilizing the cloud resources [36].

Researchers investigated speeding up regression testing through *test-suite reduction* (TSR) [18, 22, 24, 33, 34, 38, 44, 56, 66, 67, 69]. TSR identifies a subset of tests in the test suite that are redundant with respect to the remaining tests. The remaining, non-redundant tests are the *reduced test suite*, which is run instead of the original test suite for future builds [56]. Running the smaller, reduced test suite instead of the original test suite leads to faster builds in the future. While TSR can be performed manually [51], researchers have proposed many automated TSR techniques that systematically construct reduced test suites; Yoo and Harman [66] survey 83 papers on TSR published before 2011.

While TSR can provide great benefits, there are costs in using TSR. One cost is the time needed to perform TSR, but this cost is relatively small because TSR can be performed at a noncritical point of development, and the time to perform TSR is amortized over all future uses of the reduced test suite [56]. A more important cost, and the only one we consider in this paper, is that the reduced test suite may miss to detect faults in future builds that the original test suite could detect. When a build fails, the reduced test suite should ideally detect all the faults in the build (a failed build may have multiple faults) that the original test suite could detect, so the developers can debug and fix all the faults before rerunning the build. We define *Failed-Build Detection Loss* (FBDL) as the percentage of failed builds where the reduced test suite misses to detect *all* the faults that the original test suite detects. Before using TSR, the developer must decide whether the benefit from using a reduced test suite is worth the cost of missed future faults [56].

Since two decades ago, when Wong et al. [62] and Rothermel et al. [56] evaluated the fault-detection effectiveness reduction of TSR using program versions with seeded faults, all evaluations of TSR [66] (1) used seeded faults or mutants, (2) used only one fault

seeded for each faulty version, and (3) ignored new tests added to the test suite during software evolution. It is unknown how TSR performs for real software evolution. Developers have no practical guidance to make cost-benefit decisions about TSR, and researchers have no real data for improving TSR. We present the first study of TSR on real software evolution; our study (1) uses real test failures from the CI builds of open-source projects, (2) considers multiple faults per failed build, and (3) accounts for new tests added during software evolution.

More precisely, we use the following experimental procedure to evaluate the cost and benefit of TSR. First, we collect historical failed build logs for several projects and determine which tests failed in which build. Then, for a reduced tests suite computed at an earlier reduction point, we measure FBDL using the future failed builds relative to that point. One challenge in using real test failures from historical build logs is to map the test failures to the faults they detect. The mapping can range from each test failure detecting a unique fault to all test failures detecting the same single fault [56]. Prior studies do not encounter this challenge: they create faulty versions by starting from a version on which all tests pass and seeding one fault such that all test failures on the faulty version map to the one seeded fault [55]. However, we use real failed builds from CI that can encompass relatively large code changes (e.g., multiple Git commits for one push), so a faulty version (i.e., a failed build) can have multiple faults. We propose *Failure-to-Fault Map* (FFMap), a family of mappings from test failures to faults.

Our first set of experiments measure FBDL of TSR from *historical* builds. However, a developer needs the reduced test suite for *future* builds and thus needs to *predict* FBDL. Our second set of experiments measures how well the FBDL in future builds can be predicted. As predictors, we evaluate two traditional TSR metrics that can be collected at the reduction point [66], size reduction and test-requirements loss. Past studies measured test-requirements loss *implicitly* as a proxy for TSR effectiveness in the future. We are the first to *explicitly* evaluate these metrics to predict the quality of a reduced test suite for future, failed builds.

We present the first extensive study evaluating (1) the FBDL of TSR in real-world software evolution and (2) the power of various TSR metrics to predict the FBDL of TSR. No prior research has studied TSR with respect to real code changes and builds. One key challenge was to obtain a dataset with the build logs from many projects. While some companies have such logs [25, 61], most open-source projects did not have them readily available. However, increasingly many open-source projects on GitHub [7], the most popular hosting platform for open-source projects [14, 17, 19, 20], utilize Travis [11] to run tests. The usage of Travis [37] allows creating such a dataset [16]. We analyze 1478 failed builds (from a total of 27461 builds) from Travis for 32 GitHub projects written in Java and using the Maven build system. For each project, we first create reduced test suites using several TSR techniques from the literature at multiple reduction points and compute the FBDL of each reduced test suite for each mapping from FFMap. We then compute the predictor metrics for each reduced test suite and evaluate how well they predict FBDL.

The results show that FBDL is quite high, up to 52.2%, using the most "pessimistic" mapping of failures to faults from FFMap. Moreover, we find that the traditional metrics used to evaluate TSR are *not* good predictors of FBDL; the low correlation between these metrics and FBDL suggests that a developer cannot trust these metrics to predict FBDL. Because the traditional TSR metrics are not good predictors, we propose a new predictor, historical FBDL computed on historical failed build logs to predict the FBDL of future failed builds; while the historical FBDL is a better predictor than the other predictors, it is still not a good predictor.

This paper makes the following contributions:

- **Empirical Study:** We perform the first extensive study of TSR using real historical builds and test failures.
- **Novel Mappings:** We propose FFMap, a family of mappings from test failures to faults, for evaluating FBDL.
- **New Predictor:** We propose historical FBDL as a predictor of future FBDL, which performs better than traditional metrics computed for the reduced test suite.

In sum, our results confirm important concerns about TSR [56, 57, 66]. *Developers* need to exercise great caution when deciding to use TSR, because the FBDL of reduced test suites can be high, and the available predictors of FBDL are not very effective. While our proposed historical FBDL predictor performs better than the others, it is still not highly reliable. *Researchers* considering TSR need to develop (1) TSR techniques that have lower FBDL, (2) TSR techniques that result in reduced test suites that are more predictable, and/or (3) new predictors that can more reliably predict the FBDL of the reduced test suites in future builds. On a positive note, TSR has the potential to be useful. We find that a large percentage of tests never failed: performing an "oracular" TSR that is aware of all the failed tests in the future, we could have a reduced test suite consisting only of future failed tests, whose size is on average only ~20% of the original test-suite size, yet misses no regression fault in future builds.

2 EXAMPLE

We present one example that illustrates how developers would apply TSR in their project and how to evaluate the effectiveness of the reduced test suite. Consider the `caelum/vraptor4` GitHub project, "A web MVC action-based framework [...] for fast and maintainable Java development" [5]. This project uses Travis, a cloud-based continuous-integration system, to build and run tests for every push [12]. Through our large-scale study, we identified 1939 build logs for this project from Travis, and 124 of those were failed builds. Intuitively, when developers of `caelum/vraptor4` consider whether to apply TSR, they would need to consider the trade-off between the benefit of removing some number of tests and the cost of missing future faults due to removing the tests.

Assume the developer chooses the commit `b2437ab1` [1] as the reduction point for performing TSR. At this reduction point, the original test suite had 753 tests[1]. Coverage-based TSR using the Greedy algorithm [24, 40] finds that 419 of those tests are redundant and can be *removed*, so only 334 are *kept* in the reduced test suite, giving a test-suite size reduction of 44.4% (the ratio of tests from the original test suite kept in the reduced test suite).

Given a reduced test suite and its corresponding original test suite, we categorize each failed test in a future failed build based on

[1]The term *test* refers to either a test method or a test class; we discuss later how the PIT tool produces information at the level of test methods or classes.

its presence or not in the reduced test suite. Failed tests that were removed can lead to missed faults. For example, the failed build for the commit `f810dd0d` has only one failed test, but this failed test would have been removed had TSR been applied on the earlier commit `b2437ab1`, and thus the build would have not failed; in this case, TSR would have definitely been a *miss-build*, as the developer would not see *any* failure that indicates a fault. As another example, the failed build for the commit `10668287` has five failed tests, but *all* five failed tests would have been kept had TSR been applied, and thus the build not only would have still failed but also would have reported the same failed tests as when the original test suite had been run; in this case, TSR would have definitely revealed the fault.

In general, however, checking whether the failures from a reduced test suite miss a fault in a failed build is challenging because failed builds can have a mix of failed tests that are removed or kept. For example, the build number `303` [2] for the commit `021d10b7` [3] has nine failed tests, one kept and eight removed by TSR. Because one of the failed tests would have been kept, this build would have failed even if TSR had been applied. While it is positive that the build would have failed, it can be negative that the build would have had only one failed test rather than nine. In general, two or more dynamic test failures may map to either one, same static fault in the code, or they may map to several different faults. Ideally, we would like to determine whether the developers applying TSR and seeing only a subset of failures would still be able to find *all the faults*. However, it is rather challenging to determine whether multiple test failures map to the same fault.

In this particular example, our manual inspection shows that all nine failures are actually due to the same fault. The code had some encoding hardcoded to `UTF-8`, and the commit `021d10b7` [3] changed the code to get the encoding from the `web.xml` configuration file. This change broke all nine tests with a `NullPointerException`; had the developers seen any of the nine failures, they would have likely fixed their code to correct all failures. In fact, all nine tests stopped failing after just one-line change [4].

Manual inspection of failures is extremely costly and error-prone, so we use automated heuristics to categorize failed builds based on how likely developers would have fixed the code even without seeing the failures from the removed tests. We propose FFMap, a family of mappings from test failures to faults, based on the proximity of removed and kept failed tests. We describe these mappings in more detail in Section 3. For example, one of the mappings maps test failures in the same test class to the same fault. In our example, all nine test methods are from the same test class, `DefaultParametersControlTest`, and indeed fail due to the same fault. With such a mapping, we say that TSR would not miss any fault for that failed build.

3 FAILED-BUILD DETECTION LOSS (FBDL)

We describe how we measure *Failed-Build Detection Loss* (FBDL) for a reduced test suite. Given a reduced test suite from some reduction point, the goal is to find the percentage of future failed builds where the reduced test suite does not detect *all* the faults that the original test suite detected; the reduced test suite cannot detect more faults than the original test suite detects, and we do *not* assume that the original test suite detects all faults in the code. We call a failed build a *miss-build* if the reduced test suite does not detect all the faults the original test suite detects. If F is the set of all future failed builds after a reduction point, and F_r is the subset of F where the reduced test suite detects all the faults that the original test suite detects, then we define FBDL as $(|F| - |F_r|)/|F| * 100$.

Given a reduced test suite, we want to find which of the future failed builds are in F_r, using test failures from historical build logs. Wong et al. [62] and Rothermel et al. [56] defined a similar metric, but their experiments constructed faulty versions of the program, each seeded with one fault. The set of all such faulty versions is the set F, and having one fault per version makes it easy to map test failures to the faults (all test failures map to the same fault). However, in real-world evolution, a program version can have multiple faults, and mapping test failures to faults is much harder, which in turn makes defining F_r harder as well.

3.1 Failure-to-Fault Map

We develop a family of mappings from test failures to faults, called *Failure-to-Fault Map* (FFMap). The different mappings are based on heuristics for how likely certain groupings of failed tests are due to the same fault. The first mapping, FFMap$_S$, is the most "optimistic" and maps every test failure to the same fault, so any test failure detects all the faults[2]. This mapping is the same one used in seminal experiments of TSR [56, 62]. The second mapping, FFMap$_P$, maps failed tests from the same (Java) package to the same fault(s). The third mapping, FFMap$_C$, maps failed tests from the same class to the same fault(s). The final mapping, FFMap$_U$, is the most "pessimistic" mapping and maps each test failure to its own unique fault, so all test failures are needed to detect all faults. Rothermel et al. [56] also mention potentially using this mapping for TSR evaluations, but they ultimately did not use it as their experiments are such that there was one seeded fault per program version.

3.2 Classifying Failed Builds

Using the different FFMap mappings, we define which failed builds are considered a miss-build. Given a reduced test suite from some reduction point, we find every failed build after that point, classify its failed tests, and finally classify the entire failed build based on its failed tests. We assume that whenever the original test suite passed, then the reduced test suite would have also passed. This assumption can break due to test-order dependencies [15, 64] or other causes of flakiness [46, 48], but it does hold in a vast majority of cases [64].

With respect to a reduced test suite (and its corresponding original test suite) from an early, passed commit, we give a classification for each failed test from a future, failed build. Each failed test is classified as: (1) REMOVED: the test existed in the original test suite but was not in the reduced test suite; (2) KEPT: the test existed in the original test suite and was kept in the reduced test suite; (3) NEW: the test did not exist in the original test suite and is newly added between the reduction point and the failed build. We assume the test suite evolves by adding all new tests into the reduced test suite, and we consider such new test failures.

[2] When a failed build is due to only one fault, as is commonly evaluated in prior TSR studies that used one seeded fault per version, FFMap$_S$ is the correct mapping.

Table 1: Build classification based on failed tests

Failed Tests \ Build	DefMiss	In-between			Hit	NewOnly
#Removed	>0	>0			0	0
#Kept	0	>0	0	>0	>0	0
#New	0	0	>0	>0	X	>0

Based on the classification of failed tests in a failed build, we classify the failed build into one of the six classifications: DefMiss, LikelyMiss, SamePack, SameCl, Hit, and NewOnly (listed in order of "badness"). Table 1 shows how we classify entire builds based on the number of Removed, Kept, and New tests. We present the build classifications in the order that is the easiest to understand.

DefMiss Builds. A DefMiss build is one in which the reduced test suite definitely cannot detect all the fault(s) that the original test suite would detect, no matter what FFMap mapping is used. We classify a build as DefMiss if *all* of the failed tests are Removed, i.e., the only tests that fail are tests that would have been removed from the reduced test suite, so the build would have not even failed if using the reduced test suite.

Hit Builds. In contrast to a DefMiss build where the reduced test suite fails to detect any fault, a Hit build is one where the reduced test suite detects *all* the faults that the original test suite would detect, no matter what FFMap mapping is used. We classify a build as Hit if none of the failed tests are classified as Removed and at least one failed test is classified as Kept. The number of New tests does not matter.

NewOnly Builds. We classify a build as NewOnly if all the failed tests are classified as New. In these builds, neither the reduced test suite nor the original test suite detect any fault. If the reduced test suite is modified such that all new tests are added into the reduced test suite, a NewOnly build would not be a miss-build.

"In Between" Builds. When a build has a mix of Removed and Kept/New tests, it is less clear if the failed build is a miss-build or not. While the reduced test suite would have failed on the build, at least one failed test would have been removed, so we cannot easily establish whether the reduced test suite would have detected *all* the faults. These builds can be miss-builds based on which FFMap mapping is used. We classify such builds into three separate (sub)classifications. A build is SameCl if each Removed test is from the same class as some Kept/New test. The intuition is that failed tests from the same class likely detect the same fault (as illustrated in Section 2). A build is SamePack if it is not SameCl but each Removed test is from the same package as some Kept/New test. The reasoning is similar as for SameCl. All remaining builds are LikelyMiss, i.e., at least one failed Removed test does not share the same package (thus not the same class) as any Kept/New test.

3.3 Computing FBDL

With the failed builds classified, we compute which of those failed builds are miss-builds based on the FFMap mapping used. FBDL values for mappings FFMap_S, FFMap_P, FFMap_C, and FFMap_U are called FBDL_S, FBDL_P, FBDL_C, and FBDL_U, respectively.

For FBDL_S, a miss-build is a failed build classified as DefMiss, because the reduced test suite does not contain any of the failed tests in that build. For FBDL_P, a miss-build is a failed build classified as DefMiss or LikelyMiss, because the reduced test suite removed failed tests from different packages. For FBDL_C, a miss-build is a failed build classified as DefMiss, LikelyMiss, or SamePack, because the reduced test suite removed failed tests from different classes. Finally, for FBDL_U, a miss-build is a failed build classified as DefMiss, LikelyMiss, SamePack, or SameCl (i.e., only failed builds classified as Hit or NewOnly are not miss-builds), because at least one failed test is in the reduced test suite.

4 METHODOLOGY

For our evaluation of TSR, we need to obtain a dataset of projects, passed builds where we can apply TSR, and failed builds that we can use to compute the FBDL of the reduced test suites. We then evaluate if there are good predictors of FBDL. We model our experimental setup for evaluating TSR in a manner that simulates the approach by which developers could use TSR (Section 2).

4.1 Projects and Failures

To determine what projects to use for our experimental evaluation, we started with the dataset provided by TravisTorrent [16]. This dataset includes a large number of build logs harvested from Travis for a variety of projects from GitHub. We filtered the TravisTorrent dataset to obtain a set of Java projects that use Maven. We focus on projects that use Maven because we rely on the PIT mutation testing tool [10] (Section 4.3) to obtain code coverage and mutation results. As in previous research for coverage-based TSR, we use mutants to measure the loss of reduced test suites as a part of our evaluation. We also use mutants as a different test-requirement for mutant-based TSR (Section 4.3). We decided to use PIT because it is robust and increasingly used in research [21, 28, 30, 45, 58, 59].

We aim to analyze projects with a non-trivial number of failed builds, because we want a representative sample of failed builds per project. A small number of failed builds would lead to only a few possible values for FBDL (e.g., if a project has only one failed build, then the FBDL is either 0% or 100%). We found projects that have over 20 failed builds that Travis marks as either "failed" or "errored" (because both kinds can have failed tests) and at least one passed build. We considered only the builds that are either a pull request for the `master` branch or a direct push into that branch. We focused on the master branch because (1) missing failures on it is the most problematic for the project, (2) the master branch has a linear history, so we can precisely determine whether a reduced test suite from some reduction point could have been propagated to a build at another point, and (3) the master branch is more likely to have commits available for reproducing runs. We obtained 144 projects that satisfy these requirements.

4.2 Reduction Points

We attempted to rerun old builds, going back to commits from 2013. While we need not rerun the failed builds (because TSR can be evaluated from failed tests, which can be determined from logs as described in Section 4.4), we do need to rerun some passed builds to perform TSR and evaluate test-requirements loss. However, reproducing old builds (even just compiling the code) is *challenging* for many reasons (e.g., changing Java version, missing dependencies, different environment). We created a Docker [6] image similar to the one used by Travis [13]. (Travis does not make public their actual Docker image.) We aimed for multiple reduction points for

each project so we can study (1) the effects of newly added tests on the results and (2) whether the distance from the reduction point to the failed build affects the FBDL.

We first tried to reproduce a very early passed build for each project. Specifically, for each of the initially selected 144 projects, we found from the TravisTorrent dataset the earliest 10 passed builds whose commits could still be checked out from the GitHub repository. We checked out these commits and tried to reproduce the passed build by building the project in our Docker image using the commands specified in the project's `.travis.yml` file. For the earliest passed build of those 10 builds, we used its commit as a candidate point for TSR. If none of the earliest 10 builds for a project passed, we excluded that project. We were left with 51 projects.

For each of these 51 projects, we further searched for more commits on which we could reproduce passed builds to use as candidate points for TSR. Travis has many passed builds for each project. Ideally, we would evaluate TSR using all the passed builds, but given limited time and resources, we did not try them all. We searched for the builds that ran *right before* a failed build. Selecting these builds as reduction points and then evaluating FBDL on *all* the failed builds after each point gives us (1) a diverse range of commit distances from the reduction point to the failed build and (2) a diverse number of newly added tests. Specifically, for each failed build, we took up to 3 passed builds (whose commits can be checked out) right before that failed build. Each reproducible passed build provided a candidate point.

We aimed to analyze TSR for test suites with a non-trivial number of tests. We chose 10 test methods (as reported by Surefire) as the threshold. We excluded the reduction points with fewer tests, obtaining 875 candidate reduction points for 51 projects.

4.3 Test-Suite Reduction

For each commit that is a candidate reduction point, we attempted to perform TSR. We used PIT [10] to obtain coverage matrices that map individual tests to lines they cover. PIT most often reports this mapping from test *methods* to lines covered, but in some cases (e.g., when a test class has a `@BeforeClass` annotation), PIT can only map the test *classes* to lines covered (e.g., for the test class `GraphHopperServletIT` in graphhopper/graphhopper). When we refer to a *test* in a test suite, we mean either the test method or the test class that PIT reports as tests. For each project, we counted the number of these tests reported by PIT.

For some commits, PIT failed to collect coverage, either crashing altogether or having some test fail while collecting coverage even though it passed without collecting coverage (e.g., the test may be flaky [46, 48]). We filtered such commits from further analysis. Moreover, if PIT reported fewer than 10 tests in the original test suite for some commit, we filtered those commits.

For the remaining commits, we first performed coverage-based TSR. Following the traditional TSR literature [24, 66], we implemented four different TSR algorithms: Greedy [40], GE [22], GRE [23], and HGS [34]. These four algorithms are widely used in prior work on TSR [58, 59, 66, 67, 69]. All four algorithms start from an empty reduced test suite and incrementally add tests from the original test suite into the reduced test suite based on different heuristics, resulting in a (smaller) reduced test suite that still satisfies all the test-requirements satisfied by the original test suite. We applied each algorithm with the line-coverage information on each reduction point to create coverage-based reduced test suites. We excluded any reduced test suite that was the same as the original test suite. If we excluded all reduction points for a project, then we removed the project. We obtained 32 projects with 321 reduction points.

We also used PIT to collect what tests kill which mutants. We used all 16 mutation operators available in PIT, including the experimental ones [9]. Because mutation testing can be expensive, we limited PIT to run mutation testing up to 12 hours per reduction point. PIT times out or crashes on 95 reduction points, and we excluded them from any further analysis that involves mutants, which can result in excluding even entire projects from some analyses. We computed mutant-detection loss as traditionally done to evaluate coverage-based TSR at the reduction point [56, 57]. Moreover, we applied mutant-based TSR [49, 58] to construct reduced test suites that kill all mutants as the original test suites. We performed mutant-based TSR also using the same four TSR algorithms.

4.4 Extracting Failed Tests

We did not collect test failures by rerunning failed builds because rerunning old builds is challenging, as mentioned for passed builds. From the logs of failed builds, we needed to extract the names of the failed tests. For each build, the TravisTorrent dataset already provides the names of *some* failed tests extracted from the build logs. However, our sampling found that the test names in TravisTorrent were often not properly extracted. We patched the TravisTorrent analyzer for finding failed tests to extract the fully qualified test names more correctly. This extraction requires the complete textual logs for each build, and we harvested these build logs ourselves from Travis. We harvested all the logs since the first build of each project on Travis. Sometimes the build failed such that no failed test is reported in the log, e.g., the build failed in the compilation phase. For our further analysis, we only considered the failed builds where the failed test names are in the log. As a result, the analysis for some projects includes fewer than 20 failed builds.

A seemingly trivial but actually tricky aspect when extracting failed test names is to *match the names* of tests from the reduction point and the failed build. PIT provides tests that are sometimes test methods and sometimes test classes. Likewise, the reported failed tests from the Travis build logs are sometimes test methods and sometimes test classes. Our matching is as follows. If the failed test is a test method, then it matches either the exact same name or the test class of the test method in the original test suite. If the failed test is a test class, then it matches either the exact same name or *any* test method from the same class in the original test suite. If the matching finds the name, the test is Removed or Kept; otherwise, it is New. We do not consider test renames as they are not frequent in test evolution [51], and method renames are hard to track in general [52, 60]. Note that when a failed test in a future build has the same name as a test at the reduction point, the developer may have actually modified the test body between the reduction point and the failed build.

4.5 Predicting FBDL

We want to evaluate the predictive power of traditional TSR metrics that can be generated at the reduction point. We utilize a linear

Table 2: Statistics about the projects used in our evaluation; $\mu \pm \sigma$ values are across all algorithms (Greedy, GE, GRE, HGS)

ID	GitHub Slug	#Builds		Avg #Failed	#Red.	Orig.	Coverage-Based		Mutation-Based	
		Total	Failed	Test Units	Points	Size	Red. Size	Mut. Loss	Red. Size	Cov. Loss
P1	addthis/stream-lib	184	11	1.5	8	106.6	48.1% ± 1.6%	3.1% ± 0.3%	61.9% ± 1.0%	0.3% ± 0.0%
P2	azagniotov/stubby4j	886	13	3.0	9	50.1	45.2% ± 3.2%	2.8% ± 1.3%	49.2% ± 2.7%	2.5% ± 2.4%
P3	caelum/vraptor4	1939	124	4.4	7	770.6	44.1% ± 0.4%	6.5% ± 0.8%	51.1% ± 1.0%	7.1% ± 2.1%
P4	dynjs/dynjs	383	18	264.0	10	798.3	34.6% ± 0.8%	t/o	t/o	t/o
P5	FasterXML/jackson-core	580	23	2.3	20	274.4	67.5% ± 3.1%	1.8% ± 0.3%	76.9% ± 2.3%	0.3% ± 0.1%
P6	google/auto	595	33	2.8	3	27.3	65.3% ± 2.1%	0.6% ± 0.4%	63.2% ± 5.1%	8.2% ± 5.5%
P7	google/truth	419	34	10.4	3	222.7	53.3% ± 3.1%	3.3% ± 0.6%	57.0% ± 3.3%	1.6% ± 1.8%
P8	graphhopper/graphhopper	2269	117	18.3	33	686.0	23.7% ± 1.7%	t/o	t/o	t/o
P9	HubSpot/jinjava	398	6	1.7	16	307.1	41.2% ± 0.5%	4.9% ± 1.1%	53.4% ± 0.7%	1.0% ± 1.0%
P10	iluwatar/java-desig...	1537	16	2.2	1	52.0	96.2% ± 0.0%	0.8% ± 0.0%	96.2% ± 0.0%	0.9% ± 0.0%
P11	jOOQ/jOOQ	1993	88	5.8	34	29.2	69.0% ± 2.4%	0.0% ± 0.0%	69.1% ± 2.5%	0.0% ± 0.0%
P12	jsonld-java/jsonld-...	290	17	2.1	3	45.7	21.4% ± 7.9%	t/o	t/o	t/o
P13	kongchen/swagger-ma...	511	87	3.0	2	22.0	31.8% ± 0.0%	3.0% ± 0.3%	47.7% ± 2.4%	0.0% ± 0.0%
P14	ktoso/maven-git-com...	349	34	7.5	8	39.0	42.7% ± 4.3%	4.2% ± 1.3%	49.9% ± 5.3%	6.6% ± 1.7%
P15	larsga/Duke	146	15	2.6	8	640.4	34.1% ± 1.2%	6.9% ± 0.5%	48.3% ± 0.6%	0.4% ± 0.1%
P16	lviggiano/owner	582	19	4.8	25	216.7	34.5% ± 1.0%	4.5% ± 1.0%	38.0% ± 1.4%	7.0% ± 5.4%
P17	mgodave/barge	184	40	3.1	4	27.5	42.6% ± 3.4%	1.7% ± 0.2%	55.4% ± 2.5%	0.3% ± 0.6%
P18	myui/hivemall	671	50	6.9	9	85.2	57.7% ± 2.5%	t/o	t/o	t/o
P19	notnoop/java-apns	229	75	8.5	5	99.8	33.5% ± 1.0%	6.2% ± 2.1%	39.3% ± 0.6%	9.8% ± 1.8%
P20	nurkiewicz/spring-d...	101	13	53.5	1	34.0	30.9% ± 1.7%	1.4% ± 1.2%	32.4% ± 0.0%	0.0% ± 0.0%
P21	perwendel/spark	862	55	11.9	13	16.5	82.1% ± 2.5%	0.0% ± 0.0%	75.5% ± 2.0%	0.3% ± 0.1%
P22	rackerlabs/blueflood	2296	300	4.7	3	121.0	54.8% ± 1.0%	4.7% ± 0.5%	62.6% ± 0.9%	1.0% ± 0.4%
P23	redline-smalltalk/r...	228	19	2.4	3	112.3	74.5% ± 1.5%	2.9% ± 1.7%	76.8% ± 4.2%	2.2% ± 1.1%
P24	relayrides/pushy	738	30	5.6	4	47.2	66.7% ± 2.8%	0.8% ± 0.6%	68.6% ± 3.9%	0.3% ± 0.1%
P25	sanity/quickml	643	52	1.5	14	37.1	63.0% ± 6.4%	4.5% ± 2.4%	83.3% ± 2.2%	2.4% ± 1.8%
P26	scobal/seyren	453	21	13.4	3	23.3	41.0% ± 5.6%	0.9% ± 0.5%	37.5% ± 5.5%	3.4% ± 1.3%
P27	spotify/cassandra-reaper	382	21	3.7	5	20.8	68.9% ± 3.0%	0.7% ± 0.5%	77.4% ± 3.3%	0.1% ± 0.2%
P28	square/dagger	758	13	31.7	7	116.7	38.5% ± 2.3%	3.5% ± 0.7%	49.3% ± 1.9%	1.6% ± 0.6%
P29	square/wire	1404	32	11.5	2	73.0	52.5% ± 5.4%	1.1% ± 0.3%	62.0% ± 11.3%	1.4% ± 1.6%
P30	tananaev/traccar	2960	44	2.8	36	131.5	93.7% ± 2.5%	0.7% ± 0.6%	96.0% ± 1.1%	0.2% ± 0.2%
P31	twilio/twilio-java	431	13	1.5	6	88.5	73.6% ± 4.3%	0.7% ± 0.5%	72.7% ± 4.3%	1.7% ± 1.1%
P32	weld/core	2060	45	4.8	16	280.6	35.6% ± 2.2%	t/o	t/o	t/o
	Overall (Sum or Average)	27461	**1478**	504.2	321	175.1	**51.9%**	**2.7%**	**61.1%**	**2.2%**

regression model to see if there is a linear correlation between the predictor and FBDL. The linear regression model outputs an R^2 value, ranging from 0 to 1, and we are looking for an R^2 value larger than 0.7, showing strong linear correlation [31]. The p-value then shows statistical significance (the lower the better).

In case the predictor is not linearly correlated with FBDL, we also evaluate using Kendall-τ_b rank correlation, which outputs a τ_b value ranging from -1 to 1, where a negative value indicates negative correlation and a positive value indicates positive correlation. The higher the absolute value, the better the correlation, and we are again looking for an absolute value greater than 0.7, showing strong correlation [31]. There is also a p-value for statistical significance.

4.6 Summary of Analyzed Projects

Table 2 shows a summary of the 32 projects: the short ID to be used later, the GitHub user/repo "slug", the total number of builds, the number of failed builds, the average number of failed tests per failed build, the number of reduction points, the average size (i.e., the number of tests) of the test suite at the reduction points, the reduced test-suite size (relative to the original test-suite size) for coverage-based TSR using four TSR algorithms, the mutant-detection loss of the coverage-based reduced test suites, the reduced test-suite size for mutant-based TSR using four TSR algorithms, and the coverage loss of the mutant-based reduced test suites. The tabulated mean±std.dev. values are across all reduction points and all four algorithms. The cells with "t/o" mark the cases where PIT did not complete mutation testing. The overall summary numbers are: (1) the average size reduction ("Red.Size %"), 51.9% and 61.1% for coverage- and mutant-based TSR, respectively, indicate that almost *half of the tests are redundant* with respect to those criteria; and (2) the average mutant-detection loss, 2.7%, shows that, at the reduction points, coverage-based TSR results in only slightly fewer mutants killed than the original test suite; likewise, the average coverage loss, 2.2%, shows that mutant-based TSR results in very little coverage loss compared to the original test suite.

5 RESULTS AND ANALYSIS

Our evaluation aims to answer the following research questions:

RQ1 What is the FBDL of TSR in real software evaluation?
RQ2 How well can the FBDL of TSR be predicted?
RQ3 How does distance from TSR reduction point affect FBDL?

The goal of RQ1 is to measure the FBDL of TSR, which has not been done before for real-world software evolution. The goal of RQ2 is to see whether that FBDL can be predicted well: if the FBDL is high but can be predicted, the developer can still make a cost-benefit analysis about using the reduced test suite. The goal of RQ3 is related to prediction: if a longer distance from a reduction point leads to worse FBDL, the developer can make an informed decision to stop using the reduced test suite.

5.1 RQ1: FBDL

Figures 1 and 2 show $FBDL_S$, $FBDL_P$, $FBDL_C$, and $FBDL_U$ for each project using coverage-based Greedy TSR and mutant-based HGS TSR, respectively. For each project, we compute each metric for each reduction point and then average (using arithmetic mean) across all reduction points; the Avg column shows the averages across

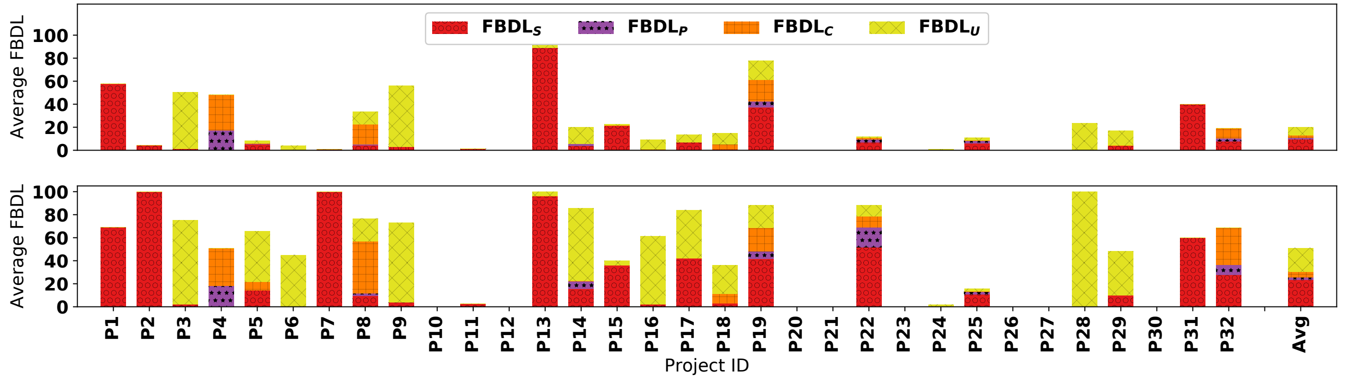

Figure 1: Average FBDL when including (top) and excluding (bottom) NewOnly (Coverage + Greedy)

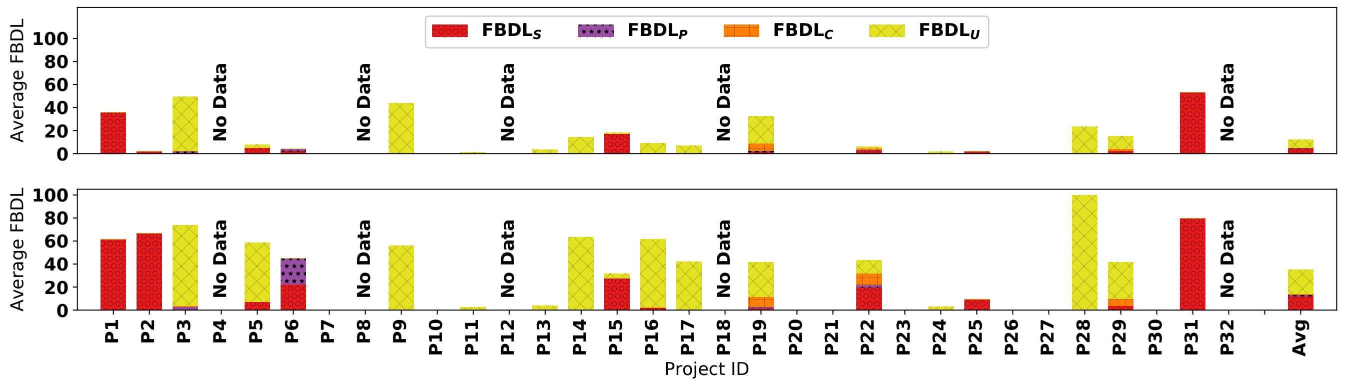

Figure 2: Average FBDL when including (top) and excluding (bottom) NewOnly (Mutants + HGS)

all projects. The bars overlap the metric values for each project, as FBDL cannot decrease going from $FBDL_S$ to $FBDL_U$. The top, respectively bottom, half of each figure shows FBDL computed including, respectively excluding, NewOnly builds.

For example, for P3 in Figure 1 the FBDL percentages of $FBDL_S$, $FBDL_P$, $FBDL_C$, and $FBDL_U$ are 1.4%, 1.4%, 1.4%, and 50.5%, respectively. Excluding the NewOnly builds, the percentages are 2.2%, 2.2%, 2.2%, and 75.2%, respectively. For this case, test-suite size reduction (average of 44.2% kept; Table 2 shows 44.1% average over all four algorithms) may be worth it as $FBDL_S$, $FBDL_P$, and $FBDL_C$ are rather low. However, if the developer believes that each test failure detects a unique fault, then based on $FBDL_U$, the reduction may not be worth it.

Figure 2 shows the breakdown per project for mutant-based TSR and has no data for five projects because PIT could not collect mutation testing results for those projects (Section 4.3). In P3, the percentages of $FBDL_S$, $FBDL_P$, $FBDL_C$, and $FBDL_U$ are 0.5%, 2.0%, 2.6%, and 49.7%, respectively. Excluding NewOnly, the percentages are 0.7%, 2.9%, 3.8%, and 73.9%, respectively. The developer may reach similar conclusions on whether or not to use the mutant-based reduced test suite for P3 as with the coverage-based one.

The two figures show the results for only one of four TSR algorithms for each criterion, coverage- and mutant-based. The distributions are visually similar for the same TSR criterion for the other three TSR algorithms (not shown for space reasons). Table 3 shows the overall averages for each FBDL for all TSR algorithms, computed excluding NewOnly builds. This table allows comparing

Table 3: Averages of FBDL for different TSR techniques

Technique		$FBDL_S$	$FBDL_P$	$FBDL_C$	$FBDL_U$
Cov	Greedy	26.1%	27.4%	28.9%	52.2%
	GE	21.3%	22.8%	29.1%	45.2%
	GRE	23.6%	24.8%	30.6%	47.2%
	HGS	22.6%	23.6%	29.1%	48.5%
Mut	Greedy	13.1%	14.7%	15.5%	36.2%
	GE	10.5%	12.0%	12.8%	34.3%
	GRE	12.2%	13.7%	14.8%	36.0%
	HGS	12.1%	13.2%	14.3%	35.5%

percentages across the two criteria, as the percentages are averaged across the builds that are *in common* between coverage- and mutant-based TSR techniques. (Recall from Section 4.3 that PIT fails to produce mutation testing results for 95 reduction points.)

We use a statistical analysis to compare each pair of algorithms for the same criterion, e.g., comparing coverage-based Greedy and coverage-based HGS. Because different FBDL are computed based on the classification of individual failed builds, we focus on comparing those classifications. Specifically, we use the Student's paired t-test to compare the ratio of builds classified in the same category, e.g., DefMiss. We find that the p-value ranges from 0.12 to 0.99 for coverage-based TSR, and from 0.25 to 0.94 for mutant-based TSR. Such high p-values fail to reject the null hypothesis that the reduced test suites from any pair of algorithms for the same criterion are from the same distribution for any failed build classification. While failing to reject the null hypothesis does not imply accepting it, all four algorithms likely behave the same *for the same TSR criterion*.

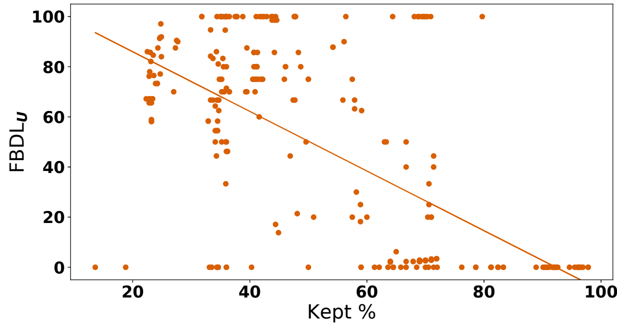

Figure 3: Size reduction vs. FBDL$_U$ (Cov Greedy)

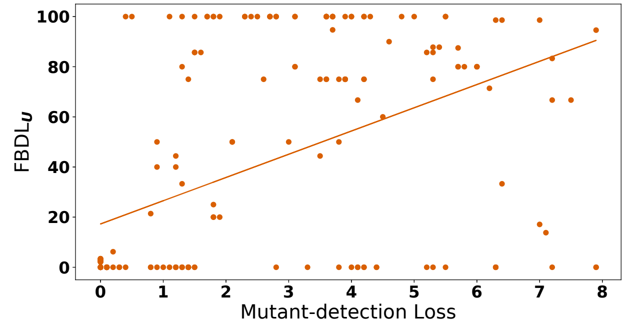

Figure 4: Mutant-detection loss vs. FBDL$_U$ (Cov Greedy)

We further compare the four TSR algorithms by viewing the classification of failed builds as a multiclass classification [8] and computing the accuracy/overlap of classifications for each pair of algorithms for each project. The average overlap across all projects and all pairs of algorithms ranges from 0.87 to 0.94 for coverage-based TSR and from 0.95 to 0.98 for mutant-based TSR. Because the overlap of failed builds that are classified the same between algorithms using the same criterion is so high, we show detailed classification results for only one representative TSR algorithm, Greedy for coverage (Figure 1) and HGS for mutants (Figure 2). Later analyses we show are only for those two TSR techniques.

We finally evaluate the correlation between pairs of FBDL variants (for the same TSR criterion and algorithm) using Kendall-τ_b rank correlation for each reduced test suite. The correlation among FBDL$_S$, FBDL$_P$, and FBDL$_C$ is rather high, with τ_b ranging from 0.69 to 0.98, all with $p < 0.0001$. However, FBDL$_U$ is not as correlated with the other FBDL values, with τ_b ranging from 0.29 to 0.80, with most less than 0.6, all with $p < 0.0001$.

A1: In sum, the FBDL cost of TSR is rather high, with a lower bound of 9.5% (based on FBDL$_S$) and an upper bound of 52.2% (based on FBDL$_U$ and excluding NewOnly builds).

5.2 RQ2: Predicting FBDL

We evaluate the predictive power of three metrics that can be measured on the reduced test suite when it is created: two metrics traditionally used for reduced test suites [66] and one new metric we propose based on historical FBDL. To focus evaluation on predicting FBDL due to tests *removed* from the original test suite, we exclude NewOnly builds; their failed tests did not exist at the reduction point and could not have been removed.

5.2.1 Test-Suite Size Reduction. Researchers commonly estimate the benefit of TSR by measuring the size of the reduced test suite relative to the size of the original test suite: the smaller the reduced test suite, the faster the build would be when using the reduced test suite. We evaluate size reduction as a predictor of FBDL. Intuitively, the more tests are kept in the reduced test suite, the less likely the reduced test suite results in a miss-build.

Figure 3 shows a scatter plot relating the size reduction and the FBDL$_U$ for each coverage-based Greedy reduced test suite. We show the prediction for FBDL$_U$ because size reduction predicts FBDL$_U$ the best. The plot includes the linear regression line. $R^2 = 0.45$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = -0.41$, with $p < 0.0001$, suggests a weak negative correlation. For the other FBDL variants, R^2 ranges from 0.04 to 0.23, all with $p < 0.001$, and τ_b ranges from -0.23 to -0.34, all with $p < 0.0001$.

There are similar trends for mutant-based TSR. For mutant-based HGS reduced test suites, we see size reduction predicts FBDL$_U$ the best; $R^2 = 0.26$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = -0.40$, with $p < 0.0001$, suggests a weak negative correlation. For the other FBDL variants, the R^2 values are the same, 0.00, but with very high p-values. The τ_b values range from -0.10 to -0.14, all with rather high p-values. Overall, these results show that size reduction at the reduction point does not correlate well with future FBDL and cannot well predict FBDL.

We also compare size reduction and FBDL *within individual projects* that have more than 5 reduced test suites (to ensure enough data points to draw any conclusions on correlations). We check if any project results in strong correlations, i.e., R^2 value or τ_b value greater than 0.7 [31]. For all coverage-based TSR, only one project, P18, results in $R^2 > 0.7$, with $p = 0.0030$, for predicting both FBDL$_S$ and FBDL$_P$. For mutant-based TSR, only one project, P5, results in $R^2 > 0.7$, with $p = 0.0002$, for predicting FBDL$_S$, FBDL$_P$, and FBDL$_C$. For τ_b values, *no* project results in an absolute value greater than 0.7, for either coverage- or mutant-based TSR. Even per project, size reduction and FBDL do not strongly correlate.

5.2.2 Test-requirements Loss. Researchers commonly measure the effectiveness of reduced test suites at the reduction point with test-requirements loss [66]. We evaluate test-requirements loss as a predictor of FBDL. We use mutant-detection loss for coverage-based TSR and coverage loss for mutant-based TSR.

Our experiments show rather low test-requirements loss, with average 2.7% across all projects for coverage-based TSR and 2.2% for mutant-based TSR; similar low percentages were reported in recent TSR studies [21, 58, 67]. Such low percentages suggest that the reduced test suites may not miss many faults. However, we find the average FBDL$_S$, the most "optimistic" FBDL metric (Section 3.3) to be much higher than the average test-requirements loss. While the test-requirements loss is not equal to FBDL, it may still be a good predictor. Intuitively, the higher the test-requirements loss, the more likely the reduced test suite results in a miss-build.

Figure 4 shows a scatter plot relating the mutant-detection loss and the FBDL$_U$ for each coverage-based Greedy reduced test suite where we could measure mutation score. We show the prediction for FBDL$_U$ because mutation-detection loss predicts FBDL$_U$ the

best. The plot includes the linear regression line. $R^2 = 0.25$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = 0.28$, with $p < 0.0001$, suggests a weak positive correlation. For the other FBDL variants, R^2 ranges from 0.02 to 0.03. For τ_b, the values are all the same, -0.04. For other variants, the p-values are all rather high.

There are similar trends for mutant-based TSR. For mutant-based HGS reduced test suites, we see coverage-loss predicts FBDL_U the best. $R^2 = 0.14$, with $p < 0.0001$, suggests a weak linear correlation; $\tau_b = 0.18$, with $p < 0.001$, suggests a weak positive correlation. For the other FBDL variants, the R^2 values are the same, 0.00, but with very high p-values. The τ_b values range from 0.11 to 0.19 (where correlating with FBDL_P has a higher τ_b value than with FBDL_U, but has lower R^2 value); the highest p-value is 0.0687.

We also analyze if test-requirements loss is a good predictor on a per-project basis, considering only projects with more than 5 reduced test suites. When correlating mutant-detection loss with FBDL, only one project, P14, has $R^2 > 0.7$, but only for predicting FBDL_P and FBDL_C, with $p = 0.0207$. Two projects have $\tau_b > 0.7$, with the highest p-value being 0.0207. When correlating coverage loss with mutant-based TSR, two projects result in $R^2 > 0.7$, with the highest p-value being 0.0570. No project has a good τ_b.

Overall, test-requirements loss is not a good predictor of FBDL for either coverage- or mutant-based TSR.

5.2.3 Combining Both Traditional Metrics.

Test-requirements loss controlled for size reduction might result in a good predictor of FBDL. We create a linear model to correlate FBDL with a linear combination of test-requirements loss and size reduction. For coverage-based Greedy reduced test suites, mutant-detection loss and size reduction together predict FBDL_U the best; $R^2 = 0.42$ (higher than before), with $p < 0.0001$. For mutant-based HGS reduced test suites, coverage loss and size reduction together predict FBDL_U the best; $R^2 = 0.27$ (higher than before), with $p < 0.0001$. The correlations are still not strong. Even a more complex model that accounts for interaction of size and test-requirements loss does not predict FBDL well.

5.2.4 Historical FBDL.

We propose to use historical FBDL to predict the future FBDL. The scenario is that a developer applies TSR at a past point and measures the FBDL that would have been obtained from that point until the current version; the developer then uses this historical FBDL to predict the FBDL of the *same* reduced test suite for future builds. (Note that this scenario does *not* re-reduce the test suite at the current point but reuses the exact same test suite from the past.) We simulate this scenario for each reduced test suite. We first find all the failed builds after the reduction point, then find the middle build among those failed builds, and finally compute the historical FBDL before the middle build and future FBDL after the middle build. We then compute correlation between historical and future FBDL.

Figure 5 shows a scatter plot relating the historical FBDL_C and the future FBDL_C for each coverage-based Greedy reduced test suite. We show the prediction for FBDL_C because it is predicted the best. The plot includes the linear regression line. $R^2 = 0.57$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = 0.64$, with $p < 0.0001$, suggests a weak positive correlation. For the other FBDL variants, R^2 ranges from 0.42 to 0.56, all with $p < 0.0001$, and τ_b ranges from

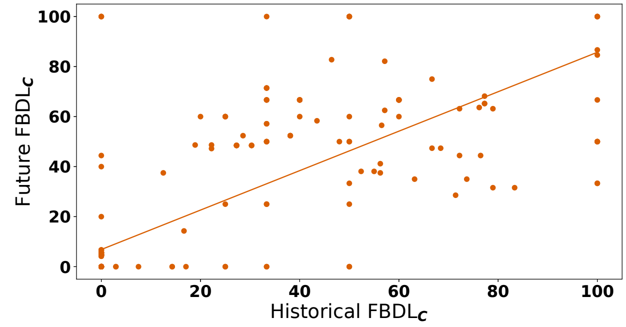

Figure 5: Historical vs. future FBDL_C (Cov Greedy)

0.40 to 0.62, all with $p < 0.0001$. For coverage-based TSR, historical FBDL is a stronger predictor than other metrics.

There are similar trends for mutant-based TSR. For mutant-based HGS reduced test suites, $R^2 = 0.55$, with $p < 0.001$, suggests a weak linear fit; $\tau_b = 0.65$, with $p < 0.001$, suggests a weak positive correlation. For the other FBDL variants, the R^2 ranges from 0.48 to 0.50, all with $p < 0.0001$, and τ_b range from 0.59 to 0.67, all with $p < 0.0001$ (FBDL_P and FBDL_C are predicted with better τ_b values than FBDL_S, but have smaller R^2 values). For mutant-based TSR, historical FBDL is again a stronger predictor than other metrics.

Per project, coverage-based Greedy reduced test suites for 6 projects achieve $R^2 > 0.7$, with the highest p-value being 0.0073; 5 projects achieve an *absolute* τ_b greater than 0.7, with the highest p-value being 0.0042. However, two of these projects have a *negative* correlation, i.e., $\tau_b < -0.7$. Mutant-based HGS reduced test suites for 4 projects have $R^2 > 0.7$, with the highest p-value 0.0038; 2 projects achieve $\tau_b > 0.7$, with the highest p-value being 0.0565. Overall, per project, historical FBDL is a better predictor of future FBDL than other metrics, though still for only a subset of projects.

A2: In sum, we find that the two traditionally used metrics (size reduction and test-requirements loss) are not good predictors of FBDL. Historical FBDL is a much better predictor of future FBDL but still not strong in most cases.

5.3 RQ3: Impact of Evolution

Although we found no good predictor for FBDL at the reduction point, some correlation may still exist between FBDL and the distance (measured in terms of the number of builds) from the reduction point to the failed builds. Intuitively, a reduced test suite results in more miss-builds at a *larger* distance from the reduction point; if so, it could lead to some actionable insight, e.g., the developers can switch back to the original test suite after some time.

We analyze the relationship between the number of builds since reduction and FBDL. Each project is analyzed separately. All builds for a project are split into 10 bins with about the same number of builds per bin. (Projects that have a different number of builds end up with different bin sizes.) For each bin, we calculate the ratio of miss-builds out of failed builds in that bin, excluding NewOnly builds. We also compute a best fit line through the 10 points given by the ratios. The slope of this line represents a simple measure of trend in FBDL with the number of builds since reduction. A positive slope is increasing FBDL, and a negative slope is decreasing FBDL.

One may expect the slope to be positive for most of the projects, i.e., the farther the builds are from the reduction point, the more likely to have missed builds. However, the results do not show this to be the case. For example, evaluating FBDL_U (which is the best predicted out of all the FBDL variants) using coverage-based Greedy TSR had 9 projects with negative slopes, 8 with slope 0, and 11 with positive slopes. (4 projects did not have enough failed builds to calculate a slope.) For the projects that have a non-zero slope, we also calculated R^2 and p-values. The R^2 values have a median of 0.28 and p-values have a median of 0.1421. The generally low R^2 values and high p-values suggest that the increasing and decreasing patterns are not strong for individual projects as well. Visual inspection also showed that plots have varying patterns, not only for coverage-based Greedy TSR but for all other techniques.

A3: FBDL does not correlate well with the distances from the reduction point, showing yet again that TSR is quite unpredictable.

6 THREATS TO VALIDITY

Our study has internal, external, and construct threats to the validity of our results. Our results may not generalize beyond the projects we evaluated. To reduce this risk, we use a diverse set of projects. Our results are based on our choice of reduction points for TSR, so a different choice may lead to different FBDL values. We reduce this risk by choosing reduction points diverse in both commit distances to failed builds and numbers of newly added tests. Different TSR algorithms guided by different criteria may result in different reduced test suites. We evaluate four widely-used TSR algorithms with two widely-used criteria.

We introduce a new metric FBDL based on mapping test failures to faults. Some of the projects in our study have a small number of flaky tests [46], which may affect our study by introducing false failures; we believe that they did not affect our key findings. We have spent substantial engineering effort trying to make the runs more reproducible using Docker, starting from the Travis Docker image [13]. Test suites may have test-order dependencies [15, 32, 43, 64, 68]. TSR inherently assumes that test suites do not have test-order dependencies [43], and our experiments assume the same.

A specific question concerning our study is whether the software project history when using TSR would look as it does when it likely did not use TSR. (Only 8 of 321 reduction points had original test suites that were smaller than those in the prior reduction point, likely because developers manually removed some tests.) If developers actually kept only the reduced test suite at some point, their behavior in the future could differ from what we see in the code repository and the build logs that used the original test suite. In theory, developers could have a completely different behavior, e.g., making different code changes or testing those changes at different times. More likely, the developers could have modified the test suites differently. For example, the developers may have added more or fewer new tests than we see currently; in the limit, a novice developer may be unaware that TSR was performed and could manually write tests similar to those that were removed.

7 RELATED WORK

Test-suite reduction (TSR) is a well-studied research topic [66]. Researchers have proposed various ways to create reduced test suites [18, 21, 23, 24, 27, 29, 33, 34, 38, 41, 44, 47, 65, 66, 69] and to evaluate the effectiveness of TSR techniques [56, 57, 62, 63, 67]. All these studies used either seeded faults or mutants to evaluate fault-detection effectiveness. We find that FBDL for a reduced test suite is much higher than its mutant-detection loss. Our definition of fault-detection effectiveness follows the approach from Wong et al. [62] and Rothermel et al. [56]; while they measured the percentage of *seeded faults* detected by the original test suite that the reduced test suite does not detect, we measure the percentage of *failed builds* where the reduced test suite does not detect all the faults. Since their experiments had one seeded fault per faulty program version, their evaluation matches our FBDL_S, where all test failures are mapped to the same fault. However, we have multiple mappings from test failures to faults in FFMap, allowing us to consider multiple faults in a build at a time, which is required when using test failures from real-world software evolution.

In our prior work [58], we studied the effects of software evolution on TSR. We measured the mutant-detection loss of the reduced test suite at the early, passed version where TSR is performed and the mutant-detection loss at a future, passed version. We found that the loss remains roughly the same, indicating TSR may be predictable. In this work, we also evaluate the effects of software evolution on TSR, but not through mutant-detection loss and rather through missed failed builds based on historical project build logs. Moreover, we evaluate whether the traditional loss metrics are good *predictors* of the missed failed builds. Unlike in our prior work, we found that TSR is unpredictable and can lead to a large percentage of missed failed builds in the future. There has been other work on prediction in the context of regression testing, but for regression test selection [35, 53, 54] as opposed to TSR.

Mutation testing is commonly used in research to evaluate the quality of testing [39]. Researchers have reported strong correlations between mutants and real faults [42] and have utilized mutants to generate tests [26, 50] and evaluate testing techniques [30, 45, 58, 59]. Our work finds that mutation testing is not a good predictor of FBDL for reduced test suites.

8 CONCLUSIONS

Automated TSR is more risky than suggested by prior research. TSR was proposed over two decades ago but since its inception it has been evaluated primarily with mutants or seeded faults. We present the first study that evaluates the cost of TSR using real test failures. Our analysis shows that FBDL can go up to 52.2%, much higher than the mutant-detection loss. Developers who are considering *current* TSR techniques should use FBDL to weigh whether the reduced test-suite size warrants the risk of missing faults. Real builds, however, do have potential for safe(r) TSR, so researchers could develop *novel* TSR techniques that either miss fewer failed builds or at least provide more predictable FBDL. Researchers should use FBDL to evaluate the quality of newly proposed TSR techniques.

ACKNOWLEDGMENTS

We thank David Craig, the attendees of the Software Engineering seminar, and the anonymous reviewers for their feedback on earlier paper drafts. This work was partially supported by the NSF grants CCF-1409423, CCF-1421503, and CNS-1646305. Suleman Mahmood was partially supported by the Sohaib and Sara Abassi Fellowship.

REFERENCES

[1] [n. d.]. https://github.com/caelum/vraptor4/commit/b2437ab1.
[2] [n. d.]. https://travis-ci.org/caelum/vraptor4/builds/15235447.
[3] [n. d.]. https://github.com/caelum/vraptor4/commit/021d10b7.
[4] [n. d.]. https://github.com/caelum/vraptor4/commit/49742a2d.
[5] [n. d.]. A web MVC action-based framework. https://github.com/caelum/vraptor4.
[6] [n. d.]. Docker. https://www.docker.com/.
[7] [n. d.]. GitHub. https://github.com/.
[8] [n. d.]. Multiclass classification. https://en.wikipedia.org/wiki/Multiclass_classification.
[9] [n. d.]. PIT Mutation Operators. http://pitest.org/quickstart/mutators/.
[10] [n. d.]. Real World Mutation Testing. http://pitest.org.
[11] [n. d.]. Travis-CI. https://travis-ci.org/.
[12] [n. d.]. Travis CI caelum/vraptor4 Builds. https://travis-ci.org/caelum/vraptor4.
[13] [n. d.]. Travis Docker Image. https://hub.docker.com/r/travisci/.
[14] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *ICPC*. 182–191.
[15] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *FSE*. 770–781.
[16] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *MSR*. 447–450.
[17] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The Promises and Perils of Mining Git. In *MSR*. 1–10.
[18] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-Criteria Models for All-Uses Test Suite Reduction. In *ICSE*. 106–115.
[19] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Predicting the Popularity of GitHub Repositories. In *PROMISE*. 9:1–9:10.
[20] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. 2014. How Do Centralized and Distributed Version Control Systems Impact Software Changes?. In *ICSE*. 322–333.
[21] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How Do Assertions Impact Coverage-Based Test-Suite Reduction?. In *ICST*. 418–423.
[22] T. Y. Chen and M. F. Lau. 1995. Heuristics Towards the Optimization of the Size of a Test Suite. In *SQM*. 415–424.
[23] T. Y. Chen and M. F. Lau. 1998. A New Heuristic for Test Suite Reduction. *IST* 40, 5-6 (1998), 347–354.
[24] T. Y. Chen and M. F. Lau. 1998. A Simulation Study on Some Heuristics for Test Suite Reduction. *IST* 40, 13 (1998), 777–787.
[25] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE*. 235–245.
[26] Gordon Fraser and Andreas Zeller. 2010. Mutation-Driven Generation of Unit Tests and Oracles. In *ISSTA*. 147–158.
[27] Jingyao Geng, Zheng Li, Ruilian Zhao, and Junxia Guo. 2016. Search Based Test Suite Minimization for Fault Detection and Localization: A Co-driven Method. In *SBSE*. 34–48.
[28] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation for Developers. In *ICSE*. 72–82.
[29] Arnaud Gotlieb and Dusica Marijan. 2014. FLOWER: Optimal Test Suite Reduction As a Network Maximum Flow. In *ISSTA*. 171–180.
[30] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause Reduction for Quick Testing. In *ICST*. 243–252.
[31] J. P. Guilford. 1956. *Fundamental Statistics in Psychology and Education*.
[32] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency. In *ISSTA*. 223–233.
[33] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-Demand Test Suite Reduction. In *ICSE*. 738–748.
[34] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *TOSEM* 2, 3 (1993), 270–285.
[35] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *TSE* 27, 3 (2001), 248–263.
[36] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *ICSE*. 483–493.
[37] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *ASE*. 426–437.
[38] Dennis Jeffrey and Neelam Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *TSE* 33, 2 (2007), 108–123.
[39] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *TSE* 37, 5 (2011), 649–678.
[40] David S. Johnson. 1974. Approximation Algorithms for Combinatorial Problems. *JCSS* (1974), 256–278.
[41] James A. Jones and Mary Jean Harrold. 2001. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. In *ICSM*. 92–102.
[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *FSE*. 654–665.
[43] Wing Lam, Sai Zhang, and Michael D. Ernst. 2015. *When Tests Collide: Evaluating and Coping with the Impact of Test Dependence*. Technical Report UW-CSE-15-03-01. University of Washington, CSE.
[44] Jun-Wei Lin and Chin-Yu Huang. 2009. Analysis of Test Suite Reduction with Enhanced Tie-breaking Techniques. *IST* 51, 4 (2009), 679–690.
[45] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-world Software Evolution?. In *ICSE*. 535–546.
[46] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *FSE*. 643–653.
[47] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. 2005. Test-Suite Reduction Using Genetic Algorithm. In *APPT*. 253–262.
[48] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *ICSE-SEIP*. 233–242.
[49] A Jefferson Offutt, Jie Pan, and Jeffrey M Voas. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *ICTCS*. 111–123.
[50] Mike Papadakis and Nicos Malevris. 2010. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *ISSRE*. 121–130.
[51] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-Suite Evolution. In *FSE*. 33:1–33:11.
[52] Napol Rachatasumrit and Miryung Kim. 2012. An Empirical Investigation into the Impact of Refactoring on Regression Testing. In *ICSM*. 357–366.
[53] David S. Rosenblum and Elaine J. Weyuker. 1996. Predicting the Cost-effectiveness of Regression Testing Strategies. In *FSE*. 118–126.
[54] David S. Rosenblum and Elaine J. Weyuker. 1997. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies. *TSE* 23, 3 (1997), 146–156.
[55] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. 2002. The Impact of Test Suite Granularity on the Cost-effectiveness of Regression Testing. In *ICSE*. 130–140.
[56] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *ICSM*. 34–43.
[57] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical Studies of Test-Suite Reduction. *STVR* 12, 4 (2002), 219–249.
[58] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-offs in Test-Suite Reduction. In *FSE*. 246–256.
[59] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-Suite Reduction and Regression Test Selection. In *ESEC/FSE*. 237–247.
[60] Gustavo Soares, Bruno Catao, Catuxe Varjao, Solon Aguiar, Rohit Gheyi, and Tiago Massoni. 2011. Analyzing Refactorings on Software Repositories. In *SBES*. 164–173.
[61] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *ISSTA*. 97–106.
[62] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *ICSE*. 41–50.
[63] W. Eric Wong, Joseph R. Horgan, Aditya P. Mathur, and Alberto Pasquini. 1997. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application. In *COMPSAC*. 522–529.
[64] Jochen Wuttke, Kıvanç Muşlu, Sai Zhang, and David Notkin. 2013. *Test Dependence: Theory and Manifestation*. Technical Report UW-CSE-13-07-02. University of Washington, CSE.
[65] Shin Yoo and Mark Harman. 2007. Pareto Efficient Multi-Objective Test Case Selection. In *ISSTA*. 140–150.
[66] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012), 67–120.
[67] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *ISSRE*. 170–179.
[68] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *ISSTA*. 385–396.
[69] Hao Zhong, Lu Zhang, and Hong Mei. 2008. An Experimental Study of Four Typical Test Suite Reduction Techniques. *IST* 50, 6 (2008), 534–546.