The Impact of Flaky Tests on Historical Test Prioritization on Chrome

Emad Fallahzadeh
Department of Computer Science
and Software Engineering
Concordia University
Montréal, Québec, Canada
emad.fallahzadeh@concordia.ca

Peter C. Rigby
Department of Computer Science
and Software Engineering
Concordia University
Montréal, Québec, Canada
peter.rigby@concordia.ca

ABSTRACT

Test prioritization algorithms prioritize probable failing tests to give faster feedback to developers in case a failure occurs. Test prioritization approaches that use historical failures to run tests that have failed in the past may be susceptible to flaky tests as these tests often fail and then pass without identifying a fault. Traditionally, flaky failures like other types of failures are considered blocking, *i.e.* a test that needs to be investigated before the code can move to the next stage. However, on Google Chrome, flaky failures are non-blocking and the code still moves to the next stage in the CI pipeline. In this work, we explain the Chrome testing pipeline and classification. Then, we re-implement two important history based test prioritization algorithms and evaluate them on over 276 million test runs from the Chrome project. We apply these algorithms in two scenarios. First, we consider flaky failures as blocking and then, we use Chrome's approach and consider flaky failures as non-blocking.

Our investigation reveals that 99.58% of all failures are flaky. These types of failures are much more repetitive than non-flaky failures, and they are also well distributed over time. We conclude that the prior performance of the prioritization algorithms have been inflated by flaky failures. We release our data and scripts in our replication package [8].

ACM Reference Format:
Emad Fallahzadeh and Peter C. Rigby. 2022. The Impact of Flaky Tests on Historical Test Prioritization on Chrome. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513038

1 INTRODUCTION

In large projects, developers make many code changes every minute [9]. To make sure that these changes do not introduce a regression into the codebase, regression tests are run. In the early era of regression testing, the number of regression tests was increasing in each individual build, delaying feedback to developers. Kim and Porter [12] observed that tests that failed in the past tend to be more likely to fail in the future, which was also confirmed in the more recent articles [13, 30]. Their method is much simpler and more cost-effective in comparison with more time-consuming approaches such as code coverage techniques for test prioritization. Statistically, their approach will provide faster feedback about failures to developers and help identify bugs in a release. With the advent of Continuous Integration (CI) and monorepos, the number of changes increased dramatically. Elbaum *et al.* adapted Kim and Porter's approach to modern CI environments [7]. Instead of considering a single build, Elbaum *et al.* considered Google's test environment, where multiple builds are being tested simultaneously by multiple machines. Consequently, they implemented the test prioritization based on the historical test failure across builds.

In this paper, we address the issue of flaky tests in a large scale test environment. Flaky tests fail and pass non-deterministically, and few prior studies explicitly deal with. For example, Elbaum *et al.* [7] do not differentiate flaky vs real test failures, but mention that flaky failures should be removed when using historical test case prioritization. Machalica *et al.* [17] filter out flaky test failures by re-running tests that fail ten times. They found that test selection using historical and other factors was less effective when flaky test failures were included. Peng *et al.* [23] reran 252 failing Travis CI jobs and found a total of 29 flaky failures. On this tiny dataset, they found that historical test prioritization is negatively impacted by the existence of flaky tests. In our work, we examine 276 million tests on Chrome to understand the impact of flaky tests on test prioritization at scale.

In this research, we study the release process of Chrome project as a representative of a large scale testing system, and we describe the process from a change commit until it is landed on the Chrome main repository. We explain the complex test status classification adopted in Chrome, and the concept of test expectations and its usage in determining the final outcome of a test. We capture the results of 276 million test runs from the Chrome project to conduct our study. This dataset not only includes the immediate results of running tests, but also the final flags chosen by the Chrome testing infrastructure, such as the flaky flags. We re-implement two of the prominent historical-based test prioritization techniques as two applicable approaches in this scale to apply on our dataset to conduct the study. We select Kim and Porter's [12] approach as one of the most effective test prioritization techniques within a single build, and we refer to it as the Kimporter algorithm. We choose Elbaum's *et al.* [7] approach as state-of-the-art for test prioritization using historical failures across builds, and we refer to it as Elbaum

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.
ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513038

algorithm. We reassess test prioritization in the context of Non-Blocking Flaky Failures (NBFF) where flaky failures do not block the builds, and we compare its results with the case of Blocking Flaky Failures (BFF). As a result, we see that the performance of the prioritization algorithms is much lower in the NBFF scenario in comparison with BFF case where the results are distorted with flaky failures. We then investigate the Chrome dataset to see the difference between the rate and distribution of different types of failures. The combination of the results and the statistical data about the flaky failures convince us that the test prioritization algorithms mostly had been prioritizing flaky failures.

The major contributions of this paper are:

- **Chrome release process:** In this research, we study the release process and steps included in Chrome from committing a change to finally, landing the change on the main Chrome repository. We also investigate the complex classification system which is used to categorize test results in Chrome.
- **Dataset:** We capture 276 million test results from the Chrome project to conduct this study. In addition to the immediate results of a test such fail and pass, this dataset consists of the final verdict which is concluded by the Chrome testing infrastructure, e.g. flaky labels. We release our data and scripts in our replication package [8].
- **Simulation:** We run two prominent history-based test prioritization algorithms on the enormous Chrome dataset by considering flaky failures as blocking versus non-blocking.
- **Outcome:** We reach the following conclusions.
  (1) Historical test prioritization algorithms are largely impacted by flaky tests.
  (2) The ratio of flaky failures is much higher than non-flaky failures, and they are much more repetitive than non-flaky failures.
  (3) Flaky failures are well distributed over time.
  (4) Previous test prioritization results are probably distorted with flaky tests.

The remainder of this paper is structured as follows. Section 2 describes the Chrome release process, its test status classification, and the way we capture data. Section 3 explains the adopted test prioritization approaches in this study. Section 4 determines how we set up the parameters to run test prioritization algorithms and how the outcomes are evaluated and interpreted. Section 5 displays the results of running the algorithms on the Chrome dataset in two different scenarios. Section 6 investigates the Chrome dataset to explain the reasons for the substantial differences in the results of these scenarios. Section 8 discusses the related works to this study. Finally, Section 9 concludes the paper.

2 BACKGROUND ON DATA

The Chrome browser is a popular open-source web browser that has 63.64% of market share of all web browsers as of February 2021[2]. We choose this project in our experiment because the software development data is publicly available, but the process is similar to that used inside Google. The scale testing at Chrome is many order of magnitude larger than the Travis CI projects, which have been the subject of recent test prioritization and flaky test papers [20, 23].

2.1 Chrome Release Process

The Chrome development team follows the following process that we outline in Figure 1.

(1) A developer uploads their code change, called a Change List (CL) at Google and Chrome, to the Gerrit code review tool.
(2) Reviewers can comment and suggest changes, which may lead to one or more new versions of the patch.
(3) Once the reviewers are satisfied with the change, it will be sent to the Commit Queue (CQ) to be emulated and tested on different platforms, *i.e.* builders, by try bots that ensure the integrity of the change through pre-submit testing.
(4) If the change is approved by all the try bots, the CQ will automatically commit the change to the Chrome main repository.
(5) Once the change is committed, also landed, the Chrome Continuous Integration system groups multiple changes made to trunk and does post-submit testing.

2.2 Test Status Classification in Chrome

Chrome is a large and complex project that does not have simple binary test outcomes. Instead, the outcomes are specified based on a predetermined set of allowable "expected results" for each test. These expected results are defined by Chrome developers on the basis of their expectations of the test results on a particular platform and their interpretation of the prior results of that test.

The most common expected results for tests are described below:

- Pass: by default, tests are expected to pass.
- Skip: tests that are disabled because they are expected to fail, especially on a particular platform. For example, a test that is specific to Windows will be skipped when run for a Linux builder.
- Fail: tests that are expected to fail.
- Timeout: tests that are expected to time out without producing any results.
- Crash: tests that are expected to crash while running.

Any expected result other than Pass or Skip is a failure expectation. Failure expectations are usually used in two cases. First, when a test is developed for a particular platform like Windows, and it is expected to fail on another platform such as Linux. In this scenario, Chrome developers prevent a failure of this test on Linux to cause multiple test reruns and blocking the build. Second, it is used for the case of flaky failures. For instance, if a test produces timeout flakiness, Chrome developers might make timeout as an expected result. Consequently, if this test time out in the next builds, it does not cause multiple test reruns and build breakage. Meanwhile, other types of failures such as crash can result in multiple test reruns and build breakage.

Based on the expected results, the final test outcome is determined as follows.

- Expected: if the result of a test is the same as its expected result, it is categorized as an expected result. Expected results do not block a build, even if they are a failure.
- Flaky: if a test fails non-deterministically, it is a flaky test. A flaky test has multiple different outcomes as a result of

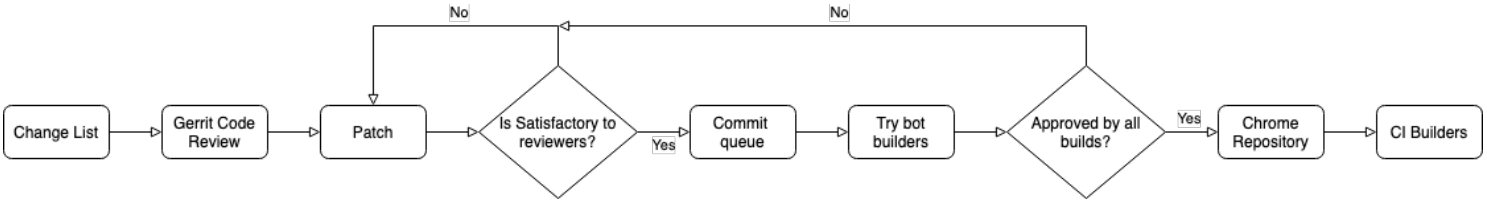

Figure 1: Chrome release process

Table 1: Final test outcomes for Chrome

Category	Actual Result
Expected	Expected failure / Pass
Flaky	Multiple different results / Unexpected pass
Unexpected	Unexpected failure

running the test multiple times on the same build. It might also lead to multiple expected results in the future.

- Unexpected: if a test fails multiple times unexpectedly, its result is considered an Unexpected outcome. This type of result will usually break the build.

Table 1 displays a summary of the final test outcomes, and Figure 2 shows a decision tree for how the decisions are made and how the final outcome is concluded based on the test results and the expected results. An allowable set of expected results is created for a test based on developer's expectations and may include: Pass, Skip, Fail, Timeout, Crash. If the outcome exists inside the list, it is considered Expected; otherwise, it is an unexpected outcome. Unexpected pass is automatically categorized as Flaky, but unexpected failure is automatically retried n times. If the result is still an unexpected failure after n retries, the test is categorized as Unexpected. If any of the retries results in an expected result, *e.g.*, a Pass, then the test failure is considered flaky and is non-blocking.

2.3 Data Processing Pipeline

Figure 3 shows the processing pipeline that we developed to gather data. It has the following steps:

(1) We called the Chrome Gerrit code review [1] APIs with a date range from 2021-01-01 to 2021-01-31 for the Chrome pre-submit testing infrastructure.
(2) We captured and stored the JSON raw data of change lists, builds, and tests.
(3) Since we are more interested in the testing results, we then filtered out the data to capture testing data for a particular builder, and it includes 276,550,812 test run verdicts for the entire period.
(4) This filtered data consists of test id, test name, test suite, build id, build start time, build end time, result id, status, final result, and test duration. We release this data in a replication package [8].

2.4 Descriptive Statistics

For the captured change lists and builds we have the following information:

- There are 9524 change lists, and an average of 307 change lists per day.
- There are 19,045 builds, and an average of 614 builds per day.
- On average, it takes about 4.06 days for a change list on the Gerrit review to be submitted to the main repository.
- There are 49,932 test suites
- There are 276,550,812 test cases for the entire period and an average of 8,920,993 per day.

3 BACKGROUND ON TEST PRIORITIZATION ALGORITHMS

Many projects deal with a large number of tests on a daily basis. To optimize the testing process, studies have proposed various test prioritization techniques. These approaches aim to increase testing performance and reduce feedback time to developers. One of the most cost-effective test prioritization methods has been the usage of historical test data. In the following, we review two historical based test prioritization techniques which are used in our experiment.

3.1 RA 1. Individual test failures within a single build

At the early stage of software development and release, the build frequency was low, but the number of tests within each build was increasing which lead to late feedback delivery to developers. As a result, some studies proposed test prioritization with the goal of prioritizing tests within each build to detect failing tests sooner [5, 28]. With this goal, some test prioritization approaches were suggested based on code coverage and the dependency of tests to the changing code [6, 26]. Meanwhile, Kim and Porter were pioneers in proposing the usage of historical test failure information to prioritize tests [12].

Since their approach does not rely on the source code dependencies, it performs more efficient than the other techniques depending on source code. The priority in the Kim and Porter algorithm is given to tests with the most recent failures.

To determine the prioritization score they adopted the exponential moving average formula from statistical quality control as in the formula 1.

$$\begin{aligned} P_0 &= 0 \\ P_k &= \alpha * h_k + (1-\alpha) * P_{k-1},\ 0 \le \alpha \le 1,\ k \ge 1 \end{aligned} \tag{1}$$

Where P_k is the probability of a test to fail at the kth observation based on its prior failures, and α is the smoothing constant adopted to weight each individual historical observation. A larger α makes recent observations more influential in the probability score. h_k

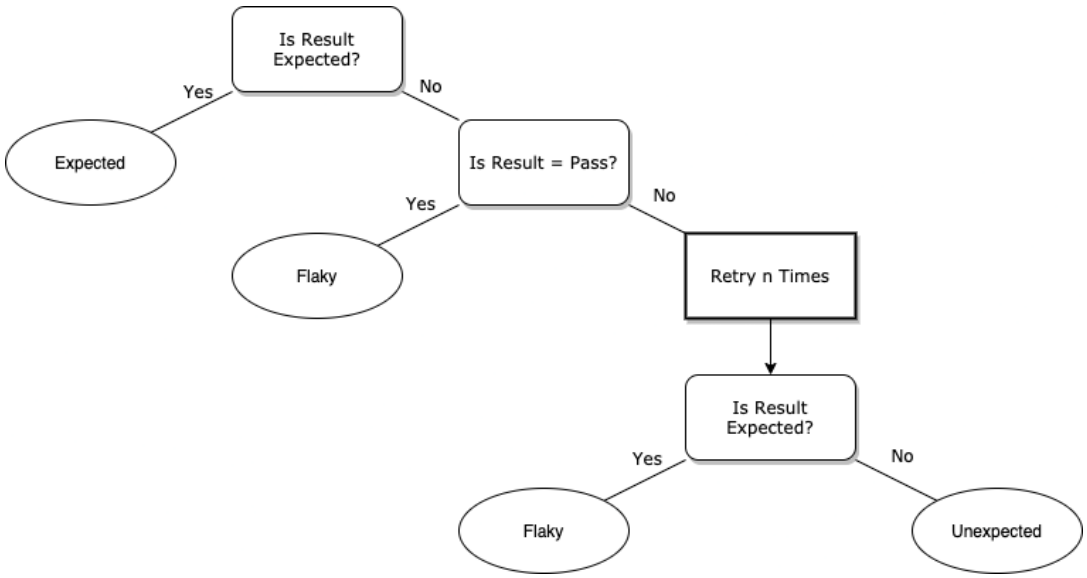

Figure 2: A decision tree for how chrome testing infrastructure decides and determines the test outcomes.

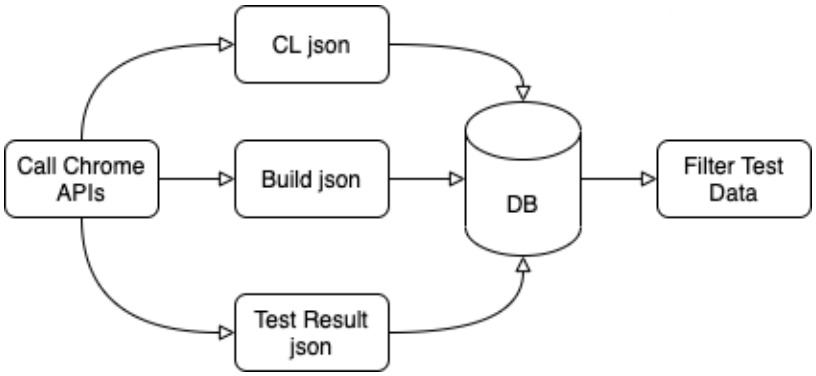

Figure 3: Data processing pipeline

shows whether the test has been failed at the kth observation, and it is assigned 1 if it failed and 0 if it passed. For the value of P_0, we followed the correction that was made by Mattis *et al.* [20] as there is no observation at P_0. We re-implement the Kim and Porter algorithm as one of the most effective test prioritization techniques, and we refer it as Kimporter algorithm in our study. We show the pseudocode implementation of Kimporter in Algorithm 1, and release our code in the replication package [8].

The Kim and Porter approach relies on the idea that tests that fail frequently in the past will fail in the future. Furthermore, usually only a few tests fail, which makes this prioritization approach even more effective. If we run this small set of frequently failing tests first, finding the first failure of a build should be more efficient.

3.2 RA 2. Individual test failures across the CI builds

Continuous integration has become a dominant release engineering practice, and the increased number of change requests has increased

Algorithm 1: Kimporter

Result: Kimporter takes tests from 1 build into the *dispatchQueue* ordering tests inside the build based on their previous failures.

```
while There are more builds do
    fill dispatchQueue with tests from 1 build;
    calculate p_k for each test inside the queue;
    prioritizeTests(dispatchQueue);
    while dispatchQueue is not empty do
        test = dispatchQueue.getNextTest();
        result = run(test);
        if result == failure then
            update h_k for the test
        end
    end
end
```

the number of builds and test runs. In a sample Google dataset, the number of test requests reaches up to 2,461 per minute [30]. As a result, they have adopted multi-machine and multi-release environment to run their enormous number of tests simultaneously [7]. This in turn has brought the concept of running and prioritizing tests *across* builds into practice.

Elbaum *et al.* introduce this challenge, and they propose a test prioritization approach which uses the same idea as Kim and Porter, but they adapt it across the builds in Google's continuous integration environment [7].

They used three time windows for implementing their algorithm. w_p as a time window for tests to be prioritized, w_f as a time window for counting the number of previous failures, and w_e as a previous execution time window to prevent low-priority tests from starvation. We re-implement their approach in Listing 2.

Algorithm 2: Elbaum

Result: Elbaum algorithm takes tests from b builds into the *dispatchQueue* and prioritizes them across the builds based on three conditions: 1. The test has recent failures 2. it is new 3. it has not been run for a long time.

```
while There are more builds do
    fill dispatchQueue with tests from the next b builds;
    foreach t_i ∈ dispatchQueue do
        if t_i ∈ w_f or t_i ∉ w_e or t_i is new then
            t_i.priority = 1
        else
            t_i.priority = 0
        end
    end
    prioritizeTests(dispatchQueue);
    while dispatchQueue is not empty do
        test = dispatchQueue.getNextTest();
        run(test);
    end
end
```

Elbaum *et al.*'s approach has the same reliance on historical test failures as the Kimporter algorithm, but it assumes multiple machines are testing builds simultaneously. The Elbaum algorithm can perform more effectively in the continuous integration environment by prioritizing failing tests across builds and running them upfront.

4 EVALUATION SETUP

To find the difference in the performance of the prioritization algorithms under study, we require metrics that can show how successful these prioritization algorithms are in finding failing tests faster in comparison to their competitor algorithms.

In this work, we are interested in evaluating the overall change in test run time at a global level where the same test may be run multiple times. Despite the dominant APFD metric [25], Elbaum *et al.* as one of the proponents of this metric for test prioritization [6] find it inappropriate to use in the CI and large scale environment [7] because it does not consider tests *across* builds. To deal with this issue, Elbaum *et al.* introduced the gain in hours for approach, *A*, relative to NoPrioritization as defined as:

$$\text{GainedHours (A)} = \text{FailTime(NoPrioritization)} - \text{FailTime}(A) \quad (2)$$

We calculate the differences between the run time of each individual failing test detected by the Kimporter, and Elbaum algorithms in comparison with NoPrioritization which is the FIFO time order of test for each build request. This will lead us to achieve the time gained for each failing test by each algorithm. To have a better insight about the distribution of the time gained by each prioritization algorithm, we use violin plots. The plot displays the median gain, first and third quartiles, outliers as well as the density distribution of data.

4.1 Evaluation Parameters

In this study, we configure our algorithms to run with the following parameters.

For the Kimporter algorithm, we consider the $\alpha = 0.8$ as in [20] to give 80% of the priority to the most recent failures. For the Elbaum algorithm, we adopt prioritization windows based on time or the number of tests [7]. We determine the prioritization windows to be a set of builds. Therefore, we set w_p to include 2 builds, w_f to consist of 48 builds, and w_e to have 96 builds. We determine these numbers relative to the time windows chosen in the original Elbaum *et al.* paper [7].

5 RESULTS AND ANALYSIS

In this section, we aim to answer the following research questions:

RQ1: What is the impact of flaky tests on history-based test prioritization algorithms?

RQ2: What is the rate of flaky failures and how repetitive are they relative to non-flaky failures?

RQ3: How are flaky and non-flaky failures distributed over time in Chrome?

5.1 RQ1: Impact of Flaky Tests on Test Prioritization

We apply the presented prioritization algorithms to the Chrome dataset in two scenarios.

First, we run the test prioritization algorithms on the 276 million test cases given the condition that flaky failures are blocking like other types of failures, and we refer it as Blocking Flaky Failures (BFF) scenario. Then, we run the same test prioritization algorithms on the same dataset considering the flaky failures as non-blocking and we refer it as Non-Blocking Flaky Failures (NBFF) case. The latter, non-blocking flaky failures, is how Chrome developers actually treat flaky failures (see Section 2.2).

Figure 4 is the violin plot showing the distribution of the GainedHours of all failing tests as a result of running the Kimporter and Elbaum test prioritization algorithms against the NoPrioritization algorithm in the BFF scenario. This plot shows the overall performance of these prioritization algorithms. The solid vertical

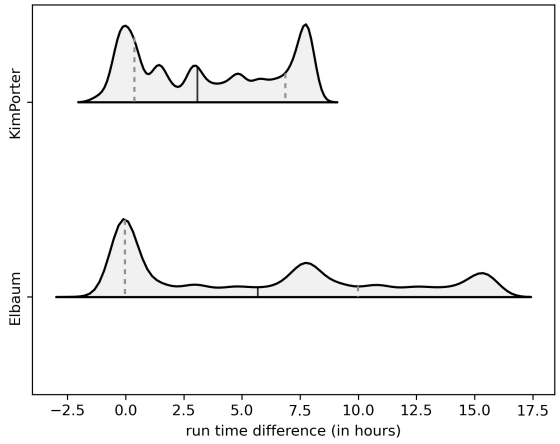

Figure 4: Violin plot showing the GainedHours of failing tests in Kimporter, and Elbaum algorithms against NoPrioritization in the BFF scenario. The solid vertical line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distribution.

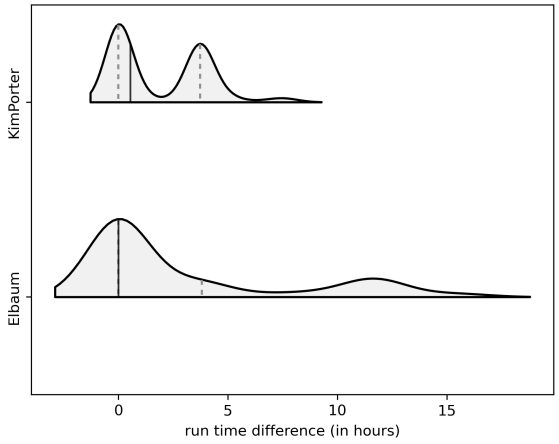

Figure 5: Violin plot showing the GainedHours of failing tests in Kimporter, and Elbaum algorithms against NoPrioritization in the NBFF scenario. The solid vertical line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distribution.

line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distributions.

As can be seen in this plot, the first quartile is at 0.37 hours for the Kimporter algorithm, which means that 75% of failing tests are detected at least 22.2 minutes faster. The median is at 3.07 hours, and the third quartile is at 6.85 hours. In the Elbaum approach, the first quartile is less than Kimporter algorithm, near zero. However, the median and the third quartile are higher than Kimporter algorithm at 5.66 hours and 10 hours respectively. In Kimporter and Elbaum algorithms there are about 18%, and 25% of failing tests that are delayed in comparison to NoPrioritization respectively. However, the median, quartiles, and most of the body of the violin plots for both Kimporter and Elbaum algorithms are on the positive sides of the distributions. This means that for most of the tests, these algorithms were able to detect the failures faster in comparison to NoPrioritization algorithm in the BFF scenario.

Figure 5 shows the violin plot distribution of the GainedHours by all failures in the NBFF scenario from running the Kimporter and Elbaum test prioritization algorithms against NoPrioritization approach. In the Kimporter approach, the first quartile is near zero which means that 25% of failures are detected by delay in comparison to NoPrioritization. The median is at 0.53 of an hour, and the third quartile is at 3.72 hours. For the Elbaum algorithm, the first quartile and the median are near zero, which means that the algorithm delayed half of the failures, and runs the other half faster. The third quartile is at 3.8 hours in GainedHours.

It is obvious in the violin plots of both Kimporter and Elbaum algorithms in the NBFF scenario that the median, quartiles, and the body of these violin plots shifted toward zero in comparison to Figure 4. This means that the performance of these prioritization algorithms deteriorated significantly when the flaky failures are considered as non-blocking failures.

Table 2: The GainedHours and the percentage of delayed failures for the Kimporter algorithm in comparison with NoPrioritization in the BFF and NBFF scenarios in 25th, 50th, and 75th percentiles.

	25th	50th	75th	% Delayed
BFF	0.37	3.07	6.86	18
NBFF	0.00	0.54	3.72	25

Table 3: The GainedHours and the percentage of delayed failures for the Elbaum algorithm in comparison with NoPrioritization in the BFF and NBFF scenarios in 25th, 50th, and 75th percentiles.

	25th	50th	75th	% Delayed
BFF	0.00	5.66	10	25
NBFF	0.00	0.00	3.8	50

Tables 2, and 3 display a summary of the GainedHours and the percentage of failures that are delayed by the Kimporter and Elbaum test prioritization algorithms against NoPrioritization approach in both BFF and NBFF scenarios.

RQ1: History-based test prioritization algorithms are ineffective when flaky tests are present.

In the following, we investigate the distribution of different types of failures in the Chrome dataset to better understand this phenomenon.

5.2 RQ2: Rate and repetitiveness of Failures

One of the important statistics we have to consider is the rate of each type of failures. For the entire period we have 276,550,812 test cases. Out of all these test cases there are 100,456 failures, and among all failures we have 416 non-flaky failures and 100,037 flaky failures. This means that the rate of non-flaky failures among failures is 0.41%, whereas the rate of flaky failures among all failures is 99.58%. It is obvious that the majority of failures are flaky failures in Chrome, *i.e.* on failure, a build passes on one of its reruns.

Another important factor we have to consider is how repetitive flaky failures are in comparison to non-flaky failures. History-based test prioritization algorithms perform rely on repetitive past failure predicting future failures. To measure the concentration of test failures, we use the Lorenz plot and Gini coefficient, which were designed to measure the concentration of wealth in a population.

Figure 6 shows the Lorenz plot for the distribution of the non-flaky failures over tests containing non-flaky failures versus the Lorenz plot for the distribution of the flaky failures over tests consisting of flaky failures and also perfect equality. The non-flaky Lorenz curve displays the cumulative growth in the number of non-flaky failures regarding the cumulative growth in the number of tests having non-flaky failures, and the flaky Lorenz curve shows the cumulative growth in the number of flaky failures considering the cumulative growth in the number of tests containing flaky failures. The straight line shows the perfect equality line, where each test passes and fails the same number of times. A larger area between a curve and the equality line results in a larger Gini coefficient value, which indicates that the distribution is more skewed. The Gini coefficient value for the non-flaky Lorenz plot is 0.57, while the Gini coefficient for the flaky Lorenz plot is 0.86.

As we can see, the Gini coefficient regarding the distribution of the flaky failures over the tests containing flaky failures is about 30% more than the Lorenz plot showing the distribution of non-flaky failures over the tests consisting of non-flaky failures. This reveals that flaky failures are much more unevenly distributed, with a smaller set of tests failing frequently. In Figure 6 we can see that about 80% of flaky failures are concentrated on only 3% of the tests containing flaky failures; whereas 80% of non-flaky failures are distributed among about 50% of tests consisting of non-flaky failures. This significant concentration of flaky failures over a minority of tests increases the repetition of these flaky failures which in turn improves the distorts the effectiveness of test prioritization algorithms that consider flaky failures.

The rate of repetition in different types of failures also confirms this phenomenon. Out of 165 tests containing non-flaky failures there are only 40 tests consisting of repetitive non-flaky failures, which stands for 24%. This low percentage of repetitive non-flaky failures can be one of the reasons for the poor performance of test prioritization algorithms in the NBFF scenario. On the other hand, from 9119 tests consisting of flaky failures about 57% are repetitive, which supports the strong performance of the prioritization algorithms in the BFF case.

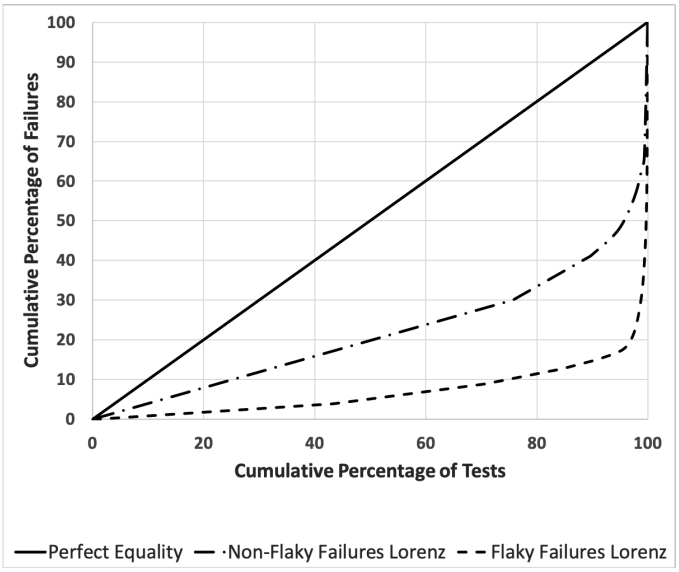

Figure 6: Lorenz plot showing the cumulative growth in the number of non-flaky failures regarding the cumulative growth in the number of tests containing non-flaky failures versus Lorenz plot displaying the cumulative growth in the number flaky failures considering the cumulative growth in the number of tests consisting of flaky failures and perfect equality on Chrome dataset.

> **RQ2:** The rate of flaky failures is much higher than non-flaky failures, and they are much more repetitive than non-flaky failures.

5.3 RQ3: Failure Distribution Over Time

One of the other aspects which matters for the performance of test prioritization algorithms is the repetitive occurrence of the failing tests over time. Since they rely on the historical failures, the more repetitive the failures are over time, there is a higher chance that the failures be detected faster.

Figure 7 displays the number of repetitive non-flaky failures for the whole month of January of 2021 on a daily basis. We can see in this chart that the number of repetitive non-flaky failures is very limited ranging from 0 to 26 daily. What is also important in this figure is that 10 days in the whole month which stands for 32% of days in the month lack repetitive non-flaky failures. The existence of very few non-flaky failures over time can dramatically impact the performance of the test prioritization algorithms.

Figure 8 shows the number of repetitive flaky failures for the whole month of January of 2021 on a daily basis. The number of repetitive flaky failures in this period ranges from 164 to 8047 per day, and there is no day without repetitive flaky failure. As we can see, in contrast to repetitive non-flaky failures, repetitive flaky failures are numerous for the whole period. This can significantly improve the performance of the history-based test prioritization algorithms.

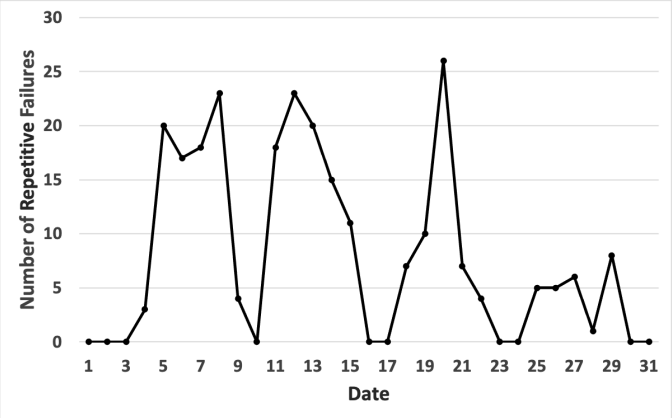

Figure 7: Time series showing the repetitive occurrence of non-flaky failures over time for the January of 2021 on a daily basis on the Chrome dataset.

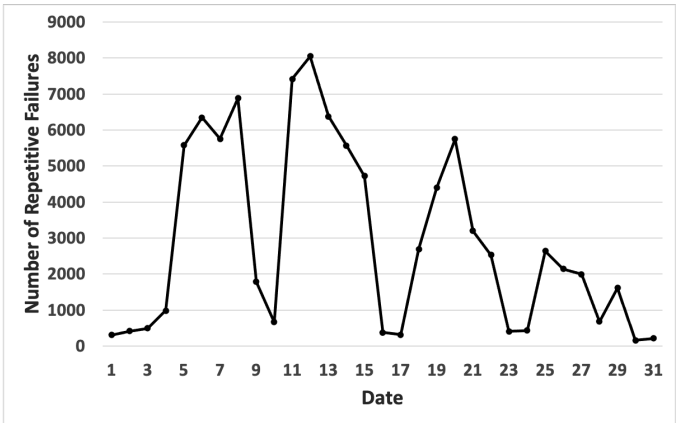

Figure 8: Time series showing the repetitive occurrence of flaky failures over time for the January of 2021 on a daily basis on the Chrome dataset.

RQ3: Flaky failures occur repetitively over time, while there are few repetitive non-flaky failures.

6 DISCUSSION

In the previous section, we saw that once the flaky failures are acknowledged as non-blocking as Chrome does, there are few non-flaky test failures, and the history based test prioritization algorithms are ineffective. We see that the Elbaum algorithm becomes ineffective with the median GainedHours of 0, and the Kimporter approach is much less effective.

One other question to answer is that whether there are any benefits in prioritizing flaky failures. Although there might not be a clear answer to this question in the literature and there is a need for further study on this matter, according to the testing pipeline in Chrome, we can say that the moment a test is detected as a flaky failure it is ignored and, it will not block the build. There are long-term procedures that Chrome testing infrastructure follows to find the cause of the flakiness and to fix it. However, there is not an immediate treatment for the flaky tests, and the goal of fast feedback might not be the case for flaky failures as they do not have any influence on the destiny of a build.

We can conclude our findings as follows. We saw that the test prioritization algorithms can become completely ineffective when flaky failures are admitted as non-blocking. We also showed that the rate of flaky failures are much higher than the non-flaky failures, and they are 99.58% of failures in Chrome. Moreover, flaky failures are much more repetitive than non-flaky failures as their failure distribution is more towards a minority of tests. Failure time series also display flaky failures as being well distributed over time, while non-flaky failures are not. All of these convince us that the test prioritization algorithms under the study had mostly prioritized flaky failures in the first place. This is due to the fact that most of the failures are flaky failures, and they have a much more repetitive nature than non-flaky failures.

The implications of this finding questions the efficacy of prior studies, that did not separate flaky failures from other types of failures, *e.g.,* Elbaum *et al.* [7] on the Google dataset.

Historical test prioritization algorithms prioritize flaky failures as they fail repetitively.

7 THREATS TO VALIDITY

7.1 External Validity

The focus of this study is on the Chrome testing data which we scraped from the Chrome website. As far as we know, there is no other publically available dataset consisting of the large number of tests as this dataset, and more importantly, having flaky labels. As a result, this dataset has been the only one we were able to work on, and we release it in our replication package for other researchers [8]. We hope developers will evaluate the impact of flaky tests failures on test case prioritization on other industrial projects.

This study concentrates on the performance of test prioritization algorithms using the historical test failures. Many other test prioritization techniques such as code coverage based techniques are not suitable and practical for the large datasets like Chrome [21]. It is also impractical to rerun tests without the massive infrastructure available to Google Chrome. We hope that future work will the impact of flaky tests on other types of test case prioritization.

7.2 Construct Validity

The metrics we use in this study are the time differences between the actual test runs in NoPrioritization order and the order suggested by our Kimporter and Elbaum algorithms. We simply plot the violin plot for each test and show the distribution. We believe that these measures are effective enough for the purpose of displaying the performance of prioritization algorithms on the Chrome dataset. We also investigated the APFD metric [25] as one of the dominant test prioritization measures; however, as Elbaum *et al.* [7] mentioned in their study, this metric does not work properly in

the continuous integration environment where the focus is on the performance of detecting failures across builds.

7.3 Internal Validity

This work shows that the test prioritization algorithms gain their performance advantage from the existence of flaky tests. Test prioritization algorithms rely on the repetitive failures over time, and this aligns well with the nature of flaky tests being repetitive. Consequently, as the results also confirm, test prioritization algorithms mostly prioritize flaky tests. We hope future work will look at other factors relevant to large-scale CI systems.

8 RELATED WORKS

Many test prioritization techniques have been proposed to optimize the regression testing process and deliver faster feedback to developers [10, 15, 16, 29]. At the early stage of regression test prioritization, the literature offers test prioritization within a single build along with the software release environment of the time. Kim and Porter [12] are the first to consider historical test failures in test prioritization. They take advantage of the exponential moving average formula to increase the weight assigned to recent test failures in their test prioritization approach. In a more recent work, Marijan *et al.* [18] explain the need for having a short feedback loop in the continuous integration development cycle. They propose a system that prioritizes tests considering the time limitation to expose the most failures.

On the other hand, big companies like Google are facing a more complex problem. They receive multiple change requests simultaneously, which requires a more advanced test prioritization. Elbaum *et al.* [7] acknowledge this problem and devise an algorithm to prioritize tests across continuous integration's builds on a sample Google dataset. They propose a waiting queue in which tests from various builds are lined up for prioritization. They adopt some time windows to control prioritization, and starvation. Most studies assume that tests are independent. However, Zhu *et al.* [30] use the previous co-failure test results to prioritize them and also re-prioritize tests once a failure occurs. They use multiple queues to control starvation and compare their algorithm with the Elbaum *et al.*'s approach by using Google and Chrome datasets. This work is based on the idea that test failures might occur together. Najafi *et al.* [22] factors in the time to run a test, longer running tests get lower priority. They design their prioritization algorithm based on the historical test failure frequency, test failure association, and the cost of the testing process. They apply their algorithm to the multi-request test environment in Ericsson. None of these prior works considered whether the failure was a flake or a true failure.

Various datasets are used for the purpose of test prioritization. Elbaum *et al.* [7] provided a sample from Google test results, which contains 3.5 million test verdicts for a period of 30 days. There are also some studies which adopted datasets taken from Travis CI to conduct their research about testing [4, 11, 20]. Recently, Matiss *et al.* [20] provided a dataset containing test results belonging to 20 Travis CI Java projects. They published their dataset to be used for test prioritization studies. Our observation shows that many studies use small projects for test optimization and flaky tests. However, the problem of test optimization and flaky tests are more relevant to the massive projects such as Chrome that have resource constraints. Another important issue is that because most of the previous studies are working on small projects they adopt approaches such as code coverage techniques that are not pragmatic in the large projects [21]. In this research, we captured and evaluated about 276 million test results from the Chrome project which presents the real testing environment in a massive project.

Flaky tests has been the subject of many studies recently [4, 14, 27]. However, there are few studies that worked on the impact of flaky tests on the release pipeline. Rahman *et al.* [24] investigate the impact of failures including flaky failures on the number of crash reports associate with Firefox builds, and they find that builds with more failures end up having higher number of crash reports. Martinez *et al.* [19] study the automatic repair system on the Defects4j project, and they reported flaky tests as one of the challenges that might mislead the automatic repair systems. Peng *et al.* [23] try to improve the IR-based test-case prioritization approaches on Travis CI projects, and they find that flaky tests can impact the test prioritization approaches. However, they do not investigate the reasons behind this impact, and the statistics and distribution of flaky tests. Moreover, most of the previous studies recognize flaky tests by re-runs. Nevertheless, they might fail to simulate the real testing environment and be subjective and unrealistic in number of re-runs and determining flaky tests. For example, Chrome usually re-runs tests twice, while FlakeFlagger baseline [3] reruns tests 1000 times. In this work, however, we are using the real flaky labels discovered by the Chrome testing infrastructure.

9 CONCLUSION

Regression testing is costly in the large continuous integration environments. To ensure appropriate scale, we captured more than 276 million test case data from the Google Chrome project, and we applied the Kimporter and Elbaum historical-based test prioritization algorithms to the dataset by first considering flaky tests as blocking, following prior work [7, 12], and then non-blocking, following Chrome's process. These algorithms rely on repetitive past failures to predict future failures. Unfortunately, flaky failures are much more common and repetitive than non-flaky failures, making these algorithms prioritize flaky tests ahead of non-flaky tests.

REFERENCES

[1] [n.d.]. Gerrit Code Review. https://chromium-review.googlesource.com
[2] [n.d.]. U.S. and global browser market share 2021. https://www.statista.com/statistics/276738/worldwide-and-us-market-share-of-leading-internet-browsers/
[3] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140 ISSN: 1558-1225.
[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 433–444. https://doi.org/10.1145/3180155.3180164 ISSN: 1558-1225.
[5] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE Computer Society, Washington, DC, USA, 329–338. http://dl.acm.org/citation.cfm?id=381473.381508 event-place: Toronto, Ontario, Canada.
[6] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00)*. Association for Computing Machinery, Portland, Oregon, USA, 102–112. https:

//doi.org/10.1145/347324.348910

[7] 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 (FSE 2014)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2635868.2635910 event-place: Hong Kong, China.

[8] Emad Fallahzadeh and Peter Rigby. 2021. Replication Package. https://github.com/CESEL/FlakyTestsInPrioritization

[9] Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of google. *Google Engineering Tools Blog* (2011).

[10] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-Box and Black-Box Test Prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 523–534. https://doi.org/10.1145/2884781.2884791 ISSN: 1558-1225.

[11] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/3238147.3238183

[12] Jung-Min Kim and A. Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 119–129. https://doi.org/10.1109/ICSE.2002.1007961

[13] 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 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 821–830. https://doi.org/10.1145/3106237.3106288

[14] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570

[15] 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 *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 535–546. https://doi.org/10.1145/2884781.2884874

[16] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 559–570. https://doi.org/10.1145/2950290.2950344

[17] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[18] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *2013 IEEE International Conference on Software Maintenance*. 540–543. https://doi.org/10.1109/ICSM.2013.91 ISSN: 1063-6773.

[19] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empir Software Eng* 22, 4 (Aug. 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[20] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3379597.3387458

[21] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[22] Armin Najafi, Weiyi Shang, and Peter C. Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 213–222. https://doi.org/10.1109/ICSE-SEIP.2019.00031

[23] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically revisiting and enhancing IR-based test-case prioritization. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 324–336. https://doi.org/10.1145/3395363.3397383

[24] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 857–862. https://doi.org/10.1145/3236024.3275529

[25] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 1999. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 179–188. https://doi.org/10.1109/ICSM.1999.792604 ISSN: 1063-6773.

[26] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct. 2001), 929–948. https://doi.org/10.1109/32.962562

[27] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: a framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 545–555. https://doi.org/10.1145/3338906.3338925

[28] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/566172.566187 event-place: Roma, Italy.

[29] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*. 192–201. https://doi.org/10.1109/ICSE.2013.6606565 ISSN: 1558-1225.

[30] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. 2018. Test Re-Prioritization in Continuous Testing Environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 69–79. https://doi.org/10.1109/ICSME.2018.00016 ISSN: 1063-6773.