Quantifying Flaky Tests to Detect Test Instabilities

Maaz Hafeez Ur Rehman

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

April 2019

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Maaz Hafeez Ur Rehman**

Entitled: **Quantifying Flaky Tests to Detect Test Instabilities**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

Dr Joey Paquet

_____ Examiner

Dr Weiyi Shang

_____ Supervisor

Dr. Peter C Rigby

Approved by _____

Dr Volker Haarslev, Graduate Program Director

15 April 2019 _____

Dr Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Quantifying Flaky Tests to Detect Test Instabilities

Maaz Hafeez Ur Rehman

A test that fails and does not lead to the identification of a fault is a flaky test. Flaky tests reduce the confidence and trust that engineers have test outcomes. For example, Google reports that 15 in 1000 test runs are flaky. Most companies rerun or remove flaky tests. Investigations of and build stops related to flaky failures are expensive. However, not all flaky tests are equal. In this work we quantify flaky tests by measuring how often they flake, *i.e.* the FlakeRate. We then define the StableFlakeRate as the number of flaky failures that a test experience in the last stable release. We compare the current FlakeRate with the StableFlakeRate to find test instabilities. We also calculate the statistical confidence testers can have in a test. Finally, we develop the BinomialStabilityOrder algorithm which we can find test instability with fewer runs.

We conduct a case study on four releases of a large project at Ericsson. We answer five research questions. We find that at Ericsson, depending on the release, between 24% to 36% have never had a flaky failure. Despite the environmental noise at Ericsson between 56% to 76% of tests have flaked less then 5/100 times and only 18% to 22% of tests have flaked more then 10/100 times. **Test confidence.** Based on the FlakeRate for each test, at the end of each release cycle, between 78% to 83% of tests have been run a sufficient number of times to be 95% confident in the test results. **Test instability.** At the end of a release cycle, between 17% to 34% of total tests will have been flagged as unstable and should be investigated by a tester. **Savings:** If testers only investigate statistically significant deviations from the StableFlakeRate, they would investigate 35% to 42% fewer tests. While stable test failures can also be investigated, priority can be given to those exhibiting instability. **Prioritizing re-runs:** We find that using only 15% of the total runs for a release, we see a decrease of 30% to 78% in ReleaseTestStability. With fewer test runs, testers can find the unstable test and investigate them earlier in the release cycle.

Acknowledgments

I would like to take this opportunity to express my gratitude for all the people who have played an indispensable role in this journey. Firstly I would like to give thanks to my supervisor Dr. Peter Rigby for the time and effort he has put forth in this thesis. His support and persistent encouragement played the most vital role in this work. In addition, I would like to thank Ericsson Inc. for providing the data and necessary hardware. My special thanks to Gary McKenna and Chris Griffiths and for their support and valuable feedback. I would also like to thank Dr. Joey Paquet and Dr. Ian Shang on my examination committee for their feedback.

I dedicate this thesis to my parents for their unconditional support for the entire duration of this work. Without their love and moral backing this work could not have been accomplished.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A test is considered flaky when it fails and there is no product fault found. This failure leads to wasted investigation by testers and a reduced confidence in the test's ability to correctly find faults. Flakiness has been found to be particularly troublesome in large continuous integration systems [39]. For example, Google estimates that 1 in 7 tests are flaky [**?**]. Google also found that when a test fails for the first time, 84% of the time it is a flaky failure that does not lead to a fault being found. Furthermore, 1.5% of test runs are flaky failures. Despite efforts to reduce the number of flaky tests, Google reports that that the number of flaky test failures remains constant.

Not all flaky tests are equal. Some tests have inherent non-determinism. For example, hardware and environmental noise can lead to test failures that do not reveal a fault. Asynchronous properties in the system under test can lead to flaky results that lead to failures that relate to synchronization issues. Furthermore, as software moves from rule based decisions to data decisions, statistical models can have a range of outputs that can make them difficult to test with deterministic tests.

In contrast, some tests that have failed to evolve with the system provide flaky results. Fixing these tests and maintaining the existing tests requires substantial costly maintenance work.

The conventional wisdom is to re-run flaky tests or to quarantine them and require developers to fix them. However, flaky tests have varying degrees of flakiness. For example, a test that has a flaky failure 1 in 100 runs is more useful than a test that has 10 flaky failures 100 runs. The goal of this work is to quantify the degree of test flakiness and to find when tests have become unstable and need investigation. In this work, we conduct a case study on the flaky tests in four releases of a large project at Ericsson. We provide answers to the following research questions.

RQ1, FlakeRate: How often does each test fail without identifying a fault?

We quantify the flakiness of tests with the FlakeRate which is the number of times a test

has failed without identifying a fault over the total number of runs. We also quantify the StableFlakeRate, which is the test flake rate of the previous stable release. By quantifying the degree of flakiness of each tests, Ericsson testers were able to examine the most flaky tests and either fix them or move them to an earlier test phase that did not involve hardware simulations.

We find that while the FlakeRate varies among releases, between 24% to 36% have never had a flaky failure. Despite the environmental noise at Ericsson between 56% to 76% of tests have flaked less then 5/100 times and only 18% to 22% of tests have flaked more then 10/100 times.

RQ2, Test Confidence: At any point in a release, how many tests have been run a sufficient number of times for testers to be confident in their outcome?

The result of a single run of a flaky tests is not reliable, and we need multiple runs to be confident in a flaky test's outcome. Based on the previous release's StableFlakeRate we use the standard statistical sample size formula to determine the proportion of tests in the current release that have been run enough times to be 95% confident in their outcome.

We find that the statistical confidence that testers can have in a test changes over a release. Based on the FlakeRate for each test, at the end of each release cycle, between 78% to 83% of tests have been run a sufficient number of times to be 95% confident in the test results.

RQ3, Test Instability: How many tests exhibit instability in the FlakeRate during a release?

We define test instability to be when the FlakeRate is different from the the StableFlakeRate at a statistically significant level. We use the binomial test to find any statistically significant deviation of the FlakeRate from the expected StableFlakeRate. The code under test and the test itself can change. If the code under test changes the FlakeRate will also change and the test will be flagged for investigation. Our goal is to detect when any modifications to the code or test occur. Test instability indicates a deviation from the expected flake rate, which indicates a potential fault in the system.

We find that when the release is rolled out, depending on the release, between 17% to 34% of total tests will have been flagged as unstable and should be investigated by a tester.

RQ4, Savings: How many test failures are likely flaky failures that can be ignored?

Without quantifying test flakiness or a mechanism to determine test stability, testers must investigate all tests that fail. However, we argue that flaky tests only need to be investigated if they deviate from their StableFlakeRate, and become unstable. Moreover testers face an overwhelming number of test fails and investigating all failures is not possible. Our goal is to find the tests that have changes by flagging them as unstable.

We find that when compared with investigating all failed tests, if testers only investigate statistically significant deviations from the StableFlakeRate, they would investigate 35% to 42% fewer tests.

While stable test failures can also be investigated, priority can be given to those exhibiting instability.

RQ5, Prioritizing re-runs: Can we prioritize test runs to find test instabilities sooner?

Flaky tests require multiple runs to determine if there has been a change in the StableFlakeRate which warrants investigation. Our goal is to look for tests with unlikely result sets and run them ahead of tests with likely result sets. We calculate the probability of a set of runs for a test using the binomial distribution. We develop the BinomialStabilityOrder algorithm to order tests based on their instability.

We find that using only 15% of the total runs for a release, we see a decrease of 30% to 78% in ReleaseTestStability. With fewer test runs, testers can find the unstable test and investigate them earlier in the release cycle.

This thesis is structured as follows. In Section 2, we broadly survey the literature on testing, flaky tests, and statistical sampling and anomaly detection. In Section 3, we formally define flaky tests and the methods we will use to differentiate and quantify them. We also introduce the Ericsson data set and simulation setup. In Section 4, we provide answers to each of our research questions. In Section 5, we discuss the feedback that we received from testers at Ericsson, the prototype tool that we released, and implications for testing at Ericsson. In Section 6, we describe our contributions and discuss future work.

Chapter 2

Background Literature

We divide the related work as follows:

- Regression testing and optimization.
- Flaky tests and their qualitative and empirical analysis.
- Sampling techniques.
- Anomaly detection techniques.

2.1 Regression Testing and Optimization

Regression testing is done when the software changes and evolves and each change is tested to determine that the system functions correctly [52]. The test cases in regression testing are designed to ensure that changes do not break the software or lead to an unanticipated or unintended behavior.

Maintenance activities can account for a large portion of the cost of software projects [49, 66]. Regression testing is an important part of software maintenance [7, 48, 66]. Prior research has established that the cost of testing can be as much as one half of the overall maintenance costs [8, 46]. This cost hinders widespread adoption of automated regression testing. In some cases, the costs of writing and running tests is as great as writing the project source code [44].

As the software evolves, test suites and constituent test cases must also evolve. Grechanik et al. [34] provide an estimate of the cost of maintaining and evolving test scripts to be $50 million to $120 million a year. The study also shows that simple changes can result in 30% to 70% of scripts needing maintenance.

When software is modified, the testers can run all test cases which is know as the retest all technique [49]. The retest all technique is expensive since the frequency of change in modern software

is high and also running all tests after a change is implemented may not be necessary. Google has reported that there are more than twenty code changes per minute across their code base and 50% of the source code files are modified on a monthly basis leading to long testing times [25, 73].

To reduce the time and cost of automated regression testing, researchers have proposed Test Suite Reduction, Regression Test Selection (*RTS*), and Test Case Prioritization (*TCP*).

Test suite reduction permanently removes the test cases that are obsolete, corrupt or unreliable to reduce the size of test suite. Reduction techniques can be very effective in reducing test suite size [37, 64, 68, 78, 79, 80]. On the other hand reduction can also compromise the fault detection capabilities of test suites [69].

RTS techniques select a subset of existing test suites to be executed. Rothermel et al. [65] presents a comprehensive survey of RTS techniques and develop a framework to evaluate and compare different RTS techniques based on metrics like inclusiveness, precision, efficiency and generality. RTS can effectively reduce the number of tests that must be run after a change in the code base [13, 33, 37, 64, 66, 14, 22, 47, 67].

Test case prioritization provides another method for reducing the cost of regression testing [70, 77]. A large body of research focuses on designing and evaluating TCP techniques [6, 16, 20, 24, 26, 27, 41, 50, 59, 63, 75, 76, 82]. Studies show that TCP can detect faults sooner than would otherwise be possible [28, 29, 71]. This allows engineers to address faults earlier.

The savings in terms of test runs and build time that come by reduction, selection and prioritization techniques do not necessarily translate into a cost effective model because each technique has extra costs. For example, RTS requires costly analysis to select tests and reduction can lead to skipped faults that would otherwise be detected by running the full test suite. Test prioritization is useful only when early detection of faults is important and it is cheaper to apply the fix at an early stage as compared to a later stage in testing. Building a cost model has to account for the savings as well as the overheads of using these techniques.

Automated regression testing is suitable only when the benefits are greater then overhead of maintaining test automation scripts. A single test fail can fail an entire build or halt the continuous integration process. Researchers have shown that tests that have failed in the past continue to fail at high levels [42]. This motivates the testers to re-order tests based on their historical likelihood to fail [30]. In rapid release model, fixed or shortened release cycles, reduce the time available for investigating all the test fails [60].

Herzig et al. [38] presents cost based test selection strategy *THEO* (Test Effectiveness Optimization Using Historic Data) that uses historic test measurements to estimate future test executions costs. The study evaluates THEO by simulating its impact on historic Windows, Office, and Dynamics development. Herzig et al. [38] states that all tests reporting failure are not reliable. The test fails

due to reliability issues are called *false alarms*. Herzig uses false alarm rate as a factor to predict the cost of running a test.

Software quality assurance is in dire need of significant progress and improvement. Software testing is resource-hungry, time-consuming and prone to human omission and error [19]. Despite huge investments in software quality assurance, code defects are routinely discovered after a release is rolled out [3]. Fixing a software product at a later stage has substantial additional cost compared to fixing it at an early stage [55]. A promising direction to reduce the bug density in production ready code is to minimize human labor using automated regression testing [17, 18, 31, 32]. However automated regression testing techniques are not ready to handle real life software with greater than a million lines of code primarily due to high resource requirements (CPU and memory).

2.2 Flaky Tests and Their Qualitative and Empirical Analysis

Flaky tests are the tests that ***PASS* or *FAIL*** in a non-deterministic way for the same code under test [10]. Due to non-deterministic behavior, flaky tests are incredibly frustrating for testers. Ideally, every new test fail should be due to the latest changes, and the developer could subsequently focus on debugging the change. However, because the outcome of flaky tests depends not only on the code, but also on tricky non-determinism (e.g. dependence on external resources or thread scheduling), they can be very difficult to debug. Moreover, if a developer does not know that a test failure is due to a flaky test rather than a regression that they introduced, how does the developer know where to start debugging: their recent changes, or the failing test? If the test failure is not due to their recent changes, then should they debug the test failure immediately, or later [10]? These issues associated with flaky tests underline the importance of developing a framework to assess their result in a reliable way.

Vahabzadeh et.al [74] conducted an extensive quantitative and qualitative study on false alarms. The researchers reported semantic errors, flaky tests, and environment-related issues as the three major causes of false alarms. Test order dependency, async wait, and concurrency are among the top causes for non-deterministic behavior [9, 61, 83, 51]. Almost all flaky tests are independent of the platform (i.e. could fail on different operating systems and hardware) [51]. In [35] is described a system to uncover the root cause of non-deterministic behavior. The technique is based on the call trace collection of flaky tests. Each call trace is linked to a pass or a fail depending on the result of execution. The call trace information is than used to construct call trees. The sub tree pattern in the call tree can lead to uncovering the reason of flakiness.

Labuschagne et al. [44] studied the cost of regression testing in practice. They found that in their

case study 18% of the total test suite executions fail. More interestingly, 13% of these failures are flaky. Of the non-flaky failures, only 74% were caused by a bug in the system under test and the remaining 26% were due to incorrect or obsolete tests. They also found that in the failed builds, only 0.38% of the test case executions find faults and 64% of failed builds containing more than one failed test.

As another example, the TAP system at google had 1.6 million test failures on average each day in the past 15 months and 73K out of the 1.6M (4.6%) of test failures are caused by flaky tests [51].

Many open source testing frameworks like android, Jenkins and Spring have annotation for flaky tests [1, 2, 4]. A study which examines flaky test in android apps concludes that some developers skip flaky tests and give up on trying to solve the problem [57]. These facts show the importance of dealing with the flaky tests to improve the quality of the regression testing.

There are multiple ways to identify flaky tests. Lou et al. [45] identifies flaky tests in their study by searching for the key-words "intermittent" and "flak" within the commit history. The study uses commit logs for identifying flaky tests that are already fixed. Eloussi et al. [45] identifies flaky tests from test results in her doctoral research where she proposes three improvements for the basic technique to identify flakiness of tests. By manually examining the flaky test, Eloussi divided the tests into three types: Non-Burstly, Burstly and State-Dependent Burstly. Another study on flaky test by Memon et al. [56] provides a detail post-mortem of flaky tests and gives actionable information about avoiding, detecting and fixing them. They inspect the test code by analyzing the code commits that likely fix flaky tests.

The most effective approach to deal with flaky tests is to run it multiple times. Another approach is to remove flaky tests from test suites or mentally ignore their results. This approach is not suitable since the testers are reluctant to remove flaky tests because they still provide coverage and find bugs [51]. This emphasizes the need to make flaky tests useful.

In [11] is described a tool called Deflaker which helps to detect and diagnose flaky test fails. The tool works by detecting the coverage of the changed code called the differential coverage. A test fail is new if the test passed on previous version of the code but fails in the current version. Deflaker tracks information about which of the tests executed any changed code. If a test had passed on previous version and now fails but has not executed any changed code than Deflaker warns it is a flaky fail. The study claims that using this approach deflaker can detect flaky tests with very low false alarm rate (1.5%) and detected 95.5% of total authentic flaky test fails as compared to only 23% detected by the rerun technique [12]. This shows the incapacity of the conventional approach to detect and deal with flaky tests by using one or two reruns. The extreme difference in the recall i.e. 95.5% and 23% by using deflaker and multiple reruns respectively is a concrete proof that the result of multiple reruns can also be flaky. Therefore, rerunning a flaky test once or twice to root out

the possibility of flaky fail is not suitable. Different flaky tests require different number of reruns depending on the false alarm rate to be able to confidently assume if the fails are flaky or not.

2.3 Statistical Sampling Techniques

In this section we present survey of sampling techniques. We use sampling in this study to indicate the number of runs required to have 95% confidence on the result of flaky tests.

Sampling is broadly divided into two main classes, quantitative and qualitative sampling [54]. The aim of quantitative sampling is to draw a representative sample from the population such that the inferences made by studying the sample can be assumed true for the population.

Miaoulis and Michener el al. [58] present the criteria influencing the sample sizes in a compact manner. According to the Miaoulis and Michener these criteria are the level of precision, the level of confidence and the variability in the attributes being measured. The level of precision is sometimes called the random sampling error and is intrinsically linked with the level of confidence. The third criterion i.e. the degree of variability in the attributes being measured refers to the distribution of the attributes in the population. Israel et al. [40] state,

> The more heterogeneous a population, the larger the sample size required to obtain a given level of precision. The less variable (more homogeneous) a population, the smaller the sample size. Note that a proportion of 50% indicates a greater level of variability than either 20% or 80%. This is because 20% and 80% indicate that a large majority do not or do, respectively, have the attribute of interest. Because a proportion of .5 indicates the maximum variability in a population, it is often used in determining a more conservative sample size, that is, the sample size may be larger than if the true variability of the population attribute were used.

Israel et al. [40] presents four different methods for obtaining a sample. These methods are using a census for a small population, using a sample size of a similar study, using published tables and using formulas to determine sample sizes. The formula approach for determining the sample sizes is suitable when different combinations of level of precision, confidence and variability are required.

Samples are divided into two categories i.e. non-probability and probability samples [40]. A probability sample is one in which every element in the population has a known probability of selection [72]. Probability samples are generally preferred over non-probability samples because the risk of incorrectly generalizing to the population can be quantified.

For large populations Cochran [23] developed the following equation to yield a sample for proportions.

$$n_0 = \frac{Z^2 pq}{e^2} \tag{1}$$

Which is valid where n_0 is the sample size, Z^2 is the abscissa of the normal curve that cuts off an area α at the tails (1 - α equals the desired confidence level, e.g., 95%), e is the desired level of precision, p is the estimated proportion of an attribute that is present in the population, and q is 1-p. The value for Z is found in statistical tables which contain the area under the normal curve. Yamane et al. [81] provides simplified formula for calculating sample sizes

$$n = \frac{N}{1 + N(e^2)} \tag{2}$$

Where n is the sample size, N is the population size, and e is the level of precision.

The use of formulas to determine sample size in the above discussion employs proportions that assume a dichotomous response for the attributes being measured. Measuring the non-deterministic behavior of flaky tests is a dichotomous response i.e. a test run result is either flaky or it is not flaky.

Kish at al. [43] says that a sample size of 30 to 200 elements is sufficient when the attribute is present 20 to 80 percent of the times.

Skewed distributions (when the attribute being measured is in very small proportions) can result in serious departures from normality even for moderate sample sizes [43]. In this case the wilson sample size formula can be employed to calculate the sample size [62].

$$n_s = z^2_{\alpha/2} \frac{\pi_0(1-\pi_0) - 2\epsilon_0^2 + \sqrt{\pi_0(1-\pi_0)^2 + 4\epsilon_0^2(\pi_0 - \frac{1}{2})^2}}{2\epsilon_0^2} \tag{3}$$

In the above equation π_0 is the FlakeRate and ϵ_0 is the margin of error.

2.4 Anomaly Detection Techniques

Anomaly detection in an important problem with application in many areas of research and industry. Anomalies are patterns or values of data that do not conform to the expected or normal behavior.

Chandola et al. [21] presents a comprehensive survey of anomaly detection techniques. The study divides anomalies into three classes i.e. point anomalies, contextual anomalies and collective anomalies. If a collection of related data is anomalous with respect to the entire data set, it is termed as collective anomaly [21]. In a collective anomaly the individual data instances may not be anomalous but their occurrence together as a collection is anomalous.

The output of any anomaly detection algorithm is either a score which gives the degree to which the instance or collection of instances is considered an anomaly or a label which simply assigns whether an instance or collection is normal or anomalous.

Anomaly detection has application in wide range of practical scenarios like intrusion detection, fraud detection, medical and public health, image processing, anomaly detection in text data and sensor networks [21].

In the software domain Hangal and Lam et al. [36] use automatic anomaly detection to find software bugs. The study proposes a tool called *DIDUCE* which checks for anomalies in program in-variants to report software bugs. The study also assigns a certain confidence level to prioritize invariant violation. The confidence is proportional to the frequency of occurrence of an invariant anomaly.

Another study uses confidence intervals to detect data outliers [84]. The process consists of obtaining a credible sample and than constructing confidence intervals to get a range of true values. If a data-point is outside the range specified by a confidence interval, it is treated as an outlier. There are multiple ways to construct confidence intervals. The normal approximation interval and the Wilson score interval to list a few. Normal approximation confidence intervals assume approximation to normal distribution and confidence intervals based on Wilson score intervals are best adapted to scenarios where the sample size is small and/or the proportion of the attribute of interest is close to 0 or 1 [5]. Brown et al. [15] gives strong recommendation of general use of Wilson score intervals over the exact binomial test. The Wilson interval takes the following mathematical form.

$$\tilde{p} \pm \frac{z_{\alpha/2}}{\tilde{n}} \sqrt{np(1-p) + \frac{z_{\alpha/2}^2}{4}} \tag{4}$$

In the above equation $\tilde{n} = n + z_{\alpha/2}^2$ and $\tilde{p} = (y + \frac{1}{2} z_{\alpha/2}^2)/(n + z_{\alpha/2}^2)$ where n is the sample size.

Chapter 3

Definitions, Methodology, and Data

3.1 Defining Flaky Tests and Test Instability

The goal of this work is to study flaky tests and identify test instability. We begin by defining the FlakeRate which is based on how often a test has failed without reveling a product fault. Tests that have flaked at one point, have an underlying degree of normal test instability. To quantify this expected flakiness, we define the StableFlakeRate as the number of flaky tests failures that occurred in the last stable version of the software. To quantify the degree of confidence a tester can have in test outcome of flaky tests, we use the standard statistical confidence formula. To find test instability, we use the binomial test to find statistically significant changes in the StableFlakeRate to the current FlakeRate for each release. Since a single run of a flaky test will not be enough to determine its outcome, we use the binomial distribution to calculate the likelihood of a set of results to prioritize which tests to re-run. We provide formal definitions below.

3.1.1 FlakeRate

One of the goals of he goal of this work is to study flaky tests and determine the varying degrees of stability of these tests. To quantify the degree of flakiness, we define the FlakeRate as the number of test fails that do not lead to a product fault (*i.e.* a bug report). The formula below formalizes the FlakeRate. For a test, t, the FlakeRate is defined over a set of test runs, r, and the number flaky test failures that did not lead to a product fault f:

$$\textsc{FlakeRate}(f, r, t) = \frac{f}{r} \tag{5}$$

3.1.2 StableFlakeRate

The FlakeRate will change as code is modified and bugs are fixed. To define a StableFlakeRate, we select the number of flaky failures that occur after the system has been stabilized. In this work, the StableFlakeRate for the current release r, is calculated from the last 90 days (stable period of a 180 days release cycle) of testing done on the previous release $r - 1$.

$$\textsc{StableFlakeRate}(R, t) = \textsc{FlakeRate}(f, r, t) \tag{6}$$

where R is limited to stable period of the previous release *i.e.* $90 < t < 180$ days for a 180 day release cycle.

The StableFlakeRate is the benchmark flakiness of tests. The ideal approach to determine the benchmark flakiness is to run a test multiple times on a stable build and count the number of times the test fails and divide it by the total number of runs. In a stable build the tests are much more likely to fail due to flakiness rather then a bug. Microsoft uses this approach to determine the StableFlakeRate. We cannot use this approach for our case study and have to look at the test execution history data to determine the benchmark flakiness. We use the test runs of the previous release as a benchmark because this indicates how flaky a test was when the code was released. A release indicates that Ericsson decided that the code was of sufficient quality to be used in customer environments. This justifies our approach to use the test runs in the previous release to determine the StableFlakeRate.

3.1.3 Statistical Confidence

Based on the StableFlakeRate, we calculate the confidence on test runs in the current release. We use the standard sample size formula to determine how many runs, r, are necessary to attain a required level of statistical confidence.

$$r = \frac{Z^2 * \textsc{StableFlakeRate} * (1 - \textsc{StableFlakeRate})}{e^2} \tag{7}$$

Where Z is the area under the normal distribution density function and e is type-I sampling error. For 95% confidence, we set $Z = 1.65$ and $e = 0.05$.

The formula can be re-written to determine at a given point, *e.g.,* at the end of release, given the number of runs for a test, r, and the StableFlakeRate, how confident are we in the test results.

$$Z = \sqrt{\frac{re^2}{\textsc{StableFlakeRate} * (1 - \textsc{StableFlakeRate})}} \tag{8}$$

Using the standard normal distribution table, we map the Z value to the confidence level for the number of runs actually performed for each test.

3.1.4 Binomial Test to Detect Instability

Over the period of a release, we compare the $\textsc{FlakeRate}(f, r, t)$ to the $\textsc{StableFlakeRate}(R, t)$ to determine if the FlakeRate for a test varies from the StableFlakeRate. We use the one tailed binomial test to determine if the current FlakeRate is higher then the StableFlakeRate. We Use the binom.test function in R which has the following notation

binom.test(f, r, StableFlakeRate, alternative="greater")

f and r refer to the total fails and total runs.

3.1.5 Binomial distribution to calculate test outcome likelihood

Flaky tests require re-runs to determine if there has been a change in the StableFlakeRate which warrants investigation. Our goal is to look for tests with most unlikely result and rerun them ahead of tests with likely result. We use the binomial distribution to determine the likelihood of the result *i.e.* for test, t, the probability, P_t, of getting f failures with r runs given StableFlakeRate:

$$P_t(f, r, \textsc{StableFlakeRate}) = \binom{r}{f} * \textsc{StableFlakeRate}^f * (1 - \textsc{StableFlakeRate})^{r-f} \tag{9}$$

We prioritize re-runs in ascending order of P_t to ensure that the most unstable tests are re-run ahead of stable tests.

3.2 Ericsson Testing and Data

In the previous section, we defined the FlakeRate and the StableFlakeRate. We also discussed the statistical approaches that will be used to detect a change in StableFlakeRate. As a case study, we have selected the Ericsson testing data of a Radio Base Station (RBS) software. Ericsson testing is done in multiple stages of integration testing. The hierarchy of testing is functionality, performance, capacity, and stability testing. For the case study we have selected the functionality testing. Functionality testing is the first stage of integration testing. The test cases in this stage are designed to assert that the systems maintains the desired functionality after each update to the code base. Since it is the first stage of integration testing after an update, the system contains the highest

Table 1: Number of runs, tests, and days for each release

Release	Number of tests	Total Runs	Days
A	7,075	789,934	425
B	8,166	1,991,475	588
C	6,980	2,519,063	619
D	6,672	1,670,010	347

number of faults. In addition to that the test environment or the simulation environment for the testing of RBS is very complex which involves multiple hardware components mimicking the base station and User equipment and RF signaling. Due to this complexity and high presence of faults, tests fail at a very high rate. The high failure rate of tests (real fails mixed with flaky fails) provides an ideal case to study the impact of test flakiness in a system.

3.2.1 Data Summary

We study 10,383 test cases and 4 subsequent releases. The data-set contains 9.9 million total test runs spanning over 895 days of testing. Of the total 10,383 test cases, 82% have flaked at some point in their lifespan. Table 1 divides the data into releases and shows the total tests in each release, total runs performed and the length of testing done on each release

3.2.2 Required Data

In order to calculate the FlakeRate and perform analysis of tests, we extract the following information form the archived history of test run data.

- Test case name (unique identifier).
- Test date and time of the test run.
- Test run verdict.
- Release id.
- TRID or Bug report id.

The above information is extracted for each test run from the archived history. Using this information we can divide the test runs into releases using the release id and calculate the FlakeRate and the StableFlakeRate as defined in Sections 3.1.1 and 3.1.2. The verdict and the TRID fields

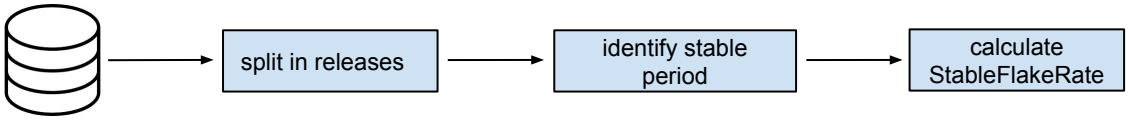

Figure 1: Obtaining StableFlakeRate

associated with each test run can be used to establish if a fail is either flaky or due to a real fault in the system. In case of a test fails due to a fault, the TRID field contains the id of the bug report. If the test fail is a flaky fail then there is no bug report id and the TRID field is empty.

3.2.3 The FlakeRate vs the StableFlakeRate

The FlakeRate is calculated for all runs in a release. When a test fails, but there is no TRID, *i.e.* bug report, filed against the test run we consider it to be a flaky failure. We divide the number of flaky fails over the total number of times at test was run for a release.

In contrast, the StableFlakeRate is calculated only with runs during the stable period. The date field in the data enables us to determine the stable period of a release. Since each release is tested for 180 days before it is rolled out, the test runs in the second half *i.e.* day 90 to 180 are defined to be the 'stable period.' Tests are less likely to fail in the stable period making this period suited to quantify the underlying degree of test flakiness

We calculate the StableFlakeRate by adding all the flaky fails and dividing by the total number of runs in the stable period. In calculating the StableFlakeRate we consider only those failed test runs which do not lead to a fault. We calculate the StableFlakeRate of each release in our case study data. The TRID or the bug report id field enables us to differentiate the test fails due to a real fault from flaky fails. If a failed test run record contains a TRID associated with it then it is an indication of a fault in the system. If there is no TRID, then the reason of failure is assumed to be the test flakiness. Using this approach we calculate the StableFlakeRate by adding all the flaky fails and dividing by the total number of runs performed in the stable period as in Equation 6

The StableFlakeRate of a release r is used in the next subsequent release $r+1$ to find instability in the test result. This allows the testers to determine if the tests in release $r+1$ have the same flakiness as expected by looking at the StableFlakeRate. We compare the cumulative FlakeRate after each run with the StableFlakeRate using the binomial test. Since the StableFlakeRate captures the underlying non-determinism of tests as seem in the previous release, we can expect the tests in the current release to flake at the same rate if there is no real fault in the system. So, if

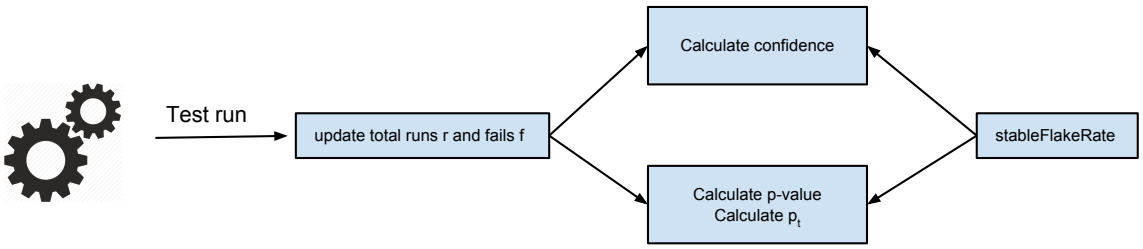

Figure 2: Comparing the FlakeRate with the StableFlakeRate

there a statistically significant negative change in the FlakeRate then it is an indication that test are failing not only due to flakiness. This allows the testers to identify test instability and filter out the test flakiness.

Chapter 4

Results

In this chapter, we answer the research questions related to the quantifying the FlakeRate, calculating the test confidence, identifying test instability, and prioritizing re-runs to tests that exhibit the highest degree of instability.

4.1 RQ1: FlakeRate

How often does each test fail without identifying a fault?

In this research question, we quantify the flakiness of tests. The common approach in industry is to classify a test as either flaky or not [39, 53**?**]. However not all flaky tests are equal. Tests with a low degree of flakiness are not as bad as the tests with a high degree of flakiness. Moreover by removing or quarantining flaky tests, parts of the system may not be tested, which could lead to bugs in the field [51]. We quantify the degree of test flakiness to understand the influence that flakiness has on a system.

We calculate the FlakeRate and the StableFlakeRate of tests for each release using Equations 5 and 6 as described in the previous section. The FlakeRate is calculated for all the runs in a release *i.e.* by adding all flaky fails of a test in a release and dividing by the total number of runs in that release. The StableFlakeRate is calculated only for the runs in the second half of the release which is considered the stable release period.

The StableFlakeRate and FlakeRate of tests for each release is plotted in Figures 3 and 4. The plots show the cumulative percentage of tests at each FlakeRate. For the StableFlakeRate from Figure 3 we observe that depending on the release between 24% and 40% of the total 10,383 tests have never had a flaky failure. Also, 38% to 60% of tests have seen less than 1/1000 flaky fails and 78% to 82% of tests have less than or equal to 1/100 flaky fails. 12% to 18% of tests have flaked more than 1/100.

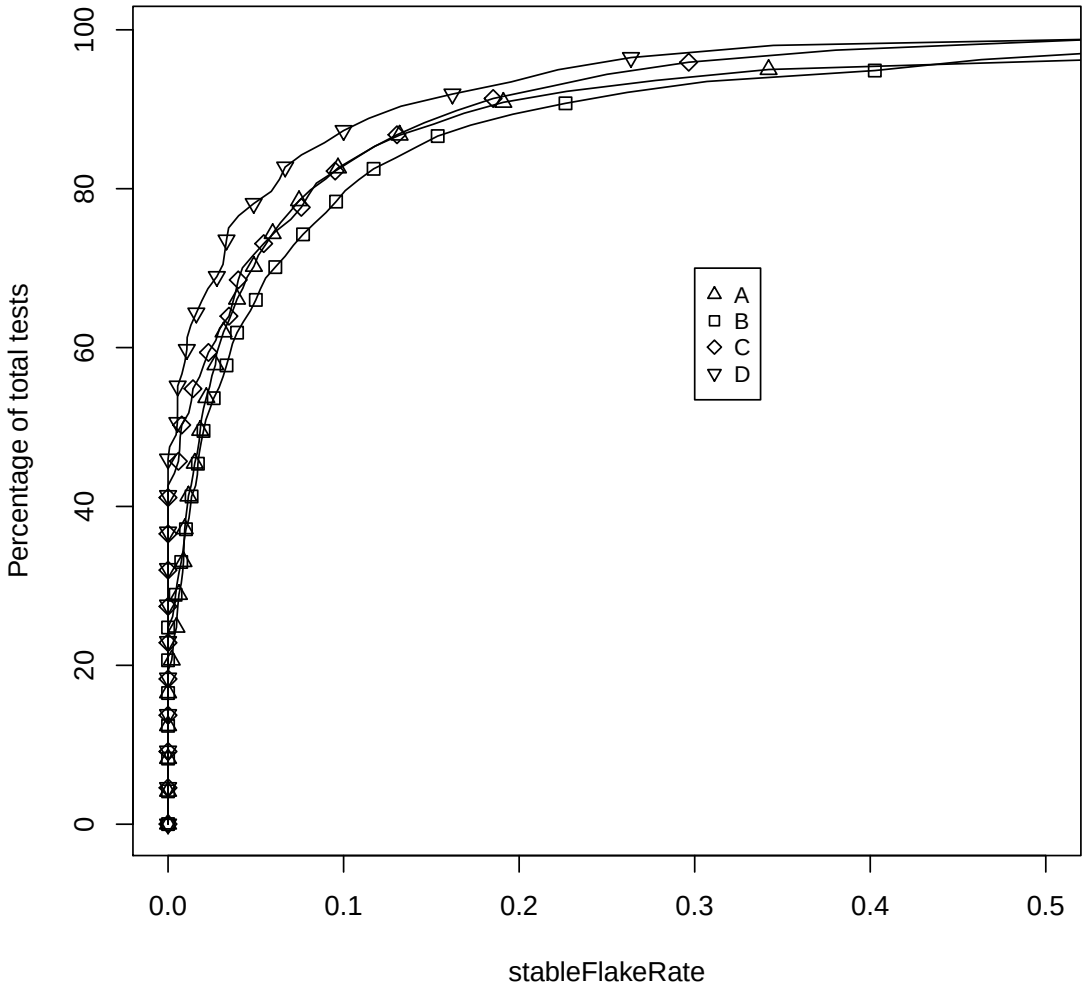

Figure 3: The cumulative density function of the StableFlakeRate for each release

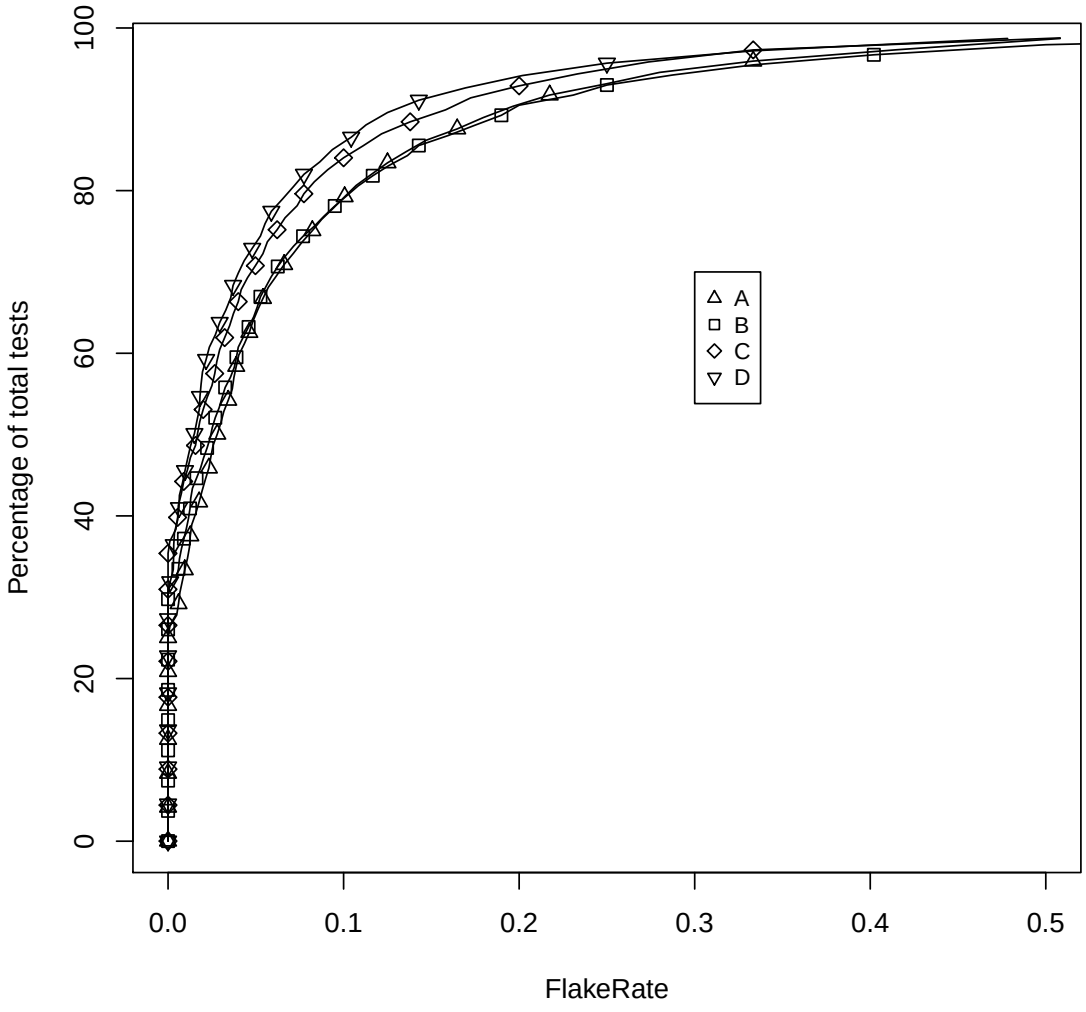

Figure 4: The cumulative density function of the FlakeRate for each release

Table 2: The quantile breakdown comparing the StableFlakeRate to the FlakeRate. The difference at the third quantile shows that fewer highly flaky tests are present in the second half of a release cycle, *i.e.* the stabilization stage of the release.

Release	StableFlakeRate			FlakeRate		
	First Qu.	Median	Third Qu.	First Qu.	Median	Third Qu.
A	0.00	0.018	0.061	0.00	0.028	0.081
B	0.00	0.020	0.079	0.00	0.024	0.08
C	0.00	0.007	0.062	0.00	0.016	0.064
D	0.0	0.004	0.034	0.0	0.014	0.053

The comparison of plots in Figures 3 and 4 also shows that StableFlakeRate is slightly lower then FlakeRate for each release. Table 2 shows the difference between the FlakeRate and the StableFlakeRate. Depending on the release the median StableFlakeRate varies between 0.004 and 0.02 whereas the median FlakeRate varies between 0.014 and 0.028. A similar pattern can be observed for the third quantile for each release. For release A and B the StableFlakeRate is 0.02 and 0.001 lower then the FlakeRate respectively. For release C and D the difference in the third quantile is 0.002 and 0.02 respectively. While clearly there are still flaky tests during the later stage of the release, the decrease in the number of highly flaky tests at the third quantile recognizes a clear decline in highly flaky tests, while a low level of flakiness, likely from the test environment is always present.

While the FlakeRate varies among releases, we see that between 24% to 36% have never had a flaky failure. Despite the environmental noise at Ericsson between 56% to 76% of tests have flaked less then 5/100 times and only 18% to 22% of tests have flaked more then 10/100 times.

4.2 RQ2: Test Confidence

At any point in a release, how many tests have been run a sufficient number of times for testers to be confident in their outcome?

For this research question, we calculate the statistical confidence of a test overtime based on the StableFlakeRate and the number of runs performed. The result of a single run of a flaky test is not reliable. We need multiple runs to be confident in flaky test's outcome. At Google a flaky test is run multiple times and reports a failure only if it fails three times in a row [**?**]. However not all flaky tests are equal, and based on the StableFlakeRate they require differing numbers of runs to attain confidence in the result. For example, using the standard statistical sampling formula in Equation 8, a test with a StableFlakeRate = 0.001 needs 1 run for 95% confidence. The corresponding values for 95% confidence for a StableFlakeRate = 0.005 is 5 runs, for StableFlakeRate = 0.01 is 10 runs, for StableFlakeRate = 0.05 is 51 runs, and for StableFlakeRate 0.1 is 95 runs.

Over each release, we calculate the number of tests that have been run enough times to be 95% confident in the result. After each test run, we increment the total runs counter r for that particular test and also f if the result is a failure. Then we use the updated total runs and the StableFlakeRate to calculate the statistical confidence. We check to see if the test reaches 95% confidence after the latest run. We plot the percentage of total tests that reach 95% confidence at daily intervals over each release. Testers can use this information to determine at any point in the release how confident they can be in their test results.

Figure 5 shows the percentage of total tests that have 95% or higher confidence over the time period for each release. Depending on the release, 20% to 38% of tests are trustworthy at 95% confidence with a single run. By day 40 we see that 50% or more of tests have reached 95% confidence for all releases. Halfway thought the release, on day 90, 68% to 80% have reached 95% confidence. The vertical line at day 180 indicates the end of testing for a release. At this point the release is rolled out. We see that 78% to 83% of total tests reach a 95% confidence by the end of a release. After the release, there are fewer test runs, with a plateau in the number of tests reaching 95% confidence.

The statistical confidence that testers can have in a test changes over a release. Based on the FlakeRate for each test, at the end of each release cycle, between 78% to 83% of tests have been run a sufficient number of times to be 95% confident in the test results.

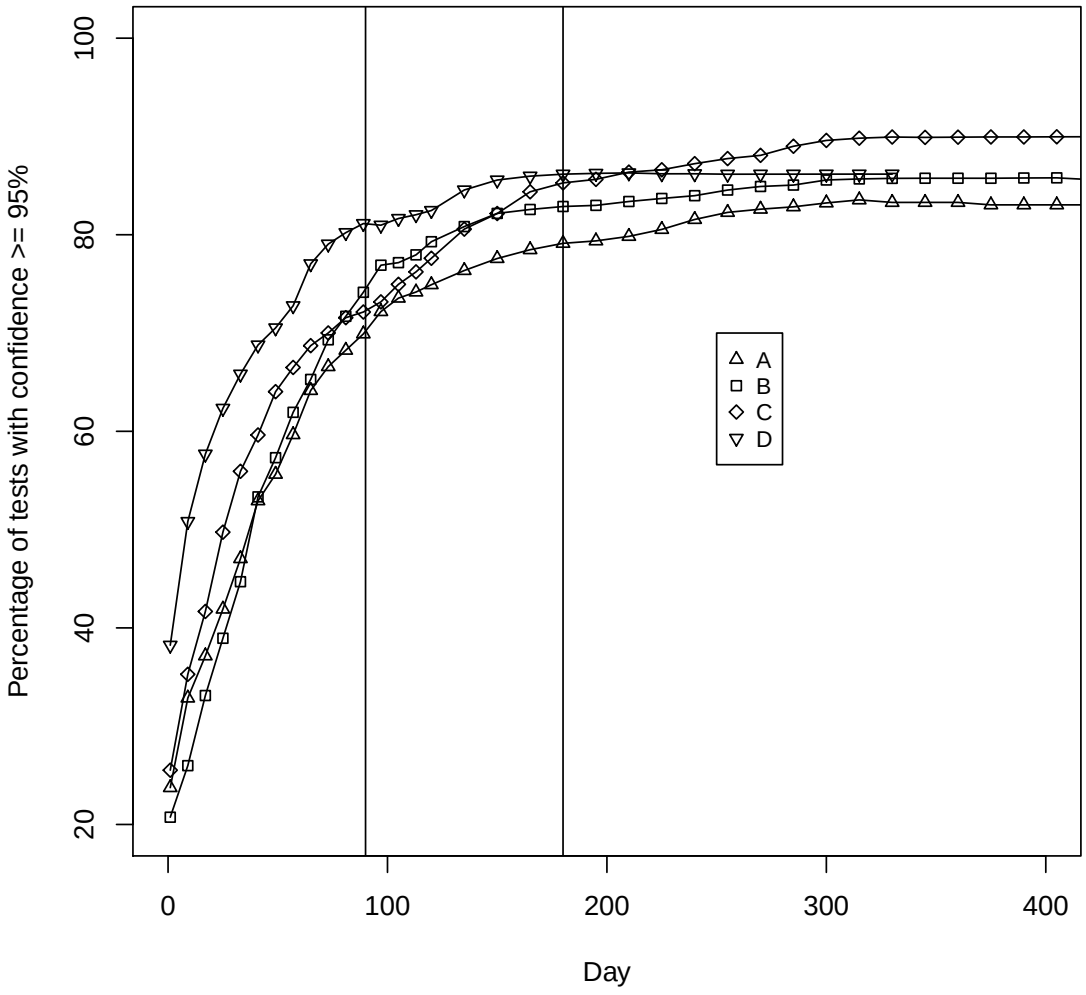

Figure 5: RQ2: The proportion of tests that reach a statistical confidence over 95% over the release cycle.

4.3 RQ3: Test Instability

How many tests exhibit instability in the FlakeRate during a release?

In the previous research question, we calculated the statistical confidence testers can have in test result over time. In this section, we measure test instability. Instability means that FlakeRate is different from the the StableFlakeRate at a statistically significant level. The code under test and the test itself can change. If the code changes the FlakeRate of the test will also change and the test will be flagged for investigation. Our goal is to find when the test is unstable and significant modification to the code and the test will be flagged for investigation. We use the binomial test, to find any statistically significant deviation of the FlakeRate from the expected StableFlakeRate distribution. Discussion with developers indicated that they are only interested in tests that have started to fail more often, *i.e.* the tests for which the FlakeRate is higher then StableFlakeRate. Using a one sided binomial test we can detect if the FlakeRate is higher the StableFlakeRate. We set the threshold of statistical significance at $p \leq 0.05$

In Figure 6, we plot the percentage of total tests that have a statistically significant negative change from StableFlakeRate at any given day for all releases. We observe variation among releases and that after 10 days, 1% to 8% of tests will have been flagged as unstable. On day 50, 10% to 17% of tests have been flagged as unstable. Halfway through the release, at day 90, between 16% and 22% are unstable. At the release date, depending on the release between 17% and 34% of total tests are unstable. After the release, at day 180, we see relatively few tests that become unstable.

For Release A, the percentage of unstable tests rises most rapidly from the start until day 20 to 14%. After day 20 until the release date there is only 1.9% increase in the tests that are unstable. For Release B the most rapid increase happens from the start until day 25 where 16.1% of tests are unstable. After day 25 there is a 6.9% increase until half way through the release at day 90 when 22% of tests are unstable. From day 90 to day 180 there is a 12% increase in percentage of tests that are unstable. For Releases C and D the percentage of unstable tests rises more steadily than for A and B. Halfway through release, C and D have 19.5% and 18.2% tests that are unstable respectively. For release D the most rapid increase is after day 118 when the unstable tests percentage rises from 19.9% to 34% (14.1% increase in 18 day). After that until day 180 there is no increase in the percentage of tests that are unstable. For Release C the unstable percentage of tests rise from 19.5% at day 90 to 27.5% at day 180.

When the release is rolled out at day 180, 17% to 34% of total tests will have been flagged as unstable and should be investigated by a tester.

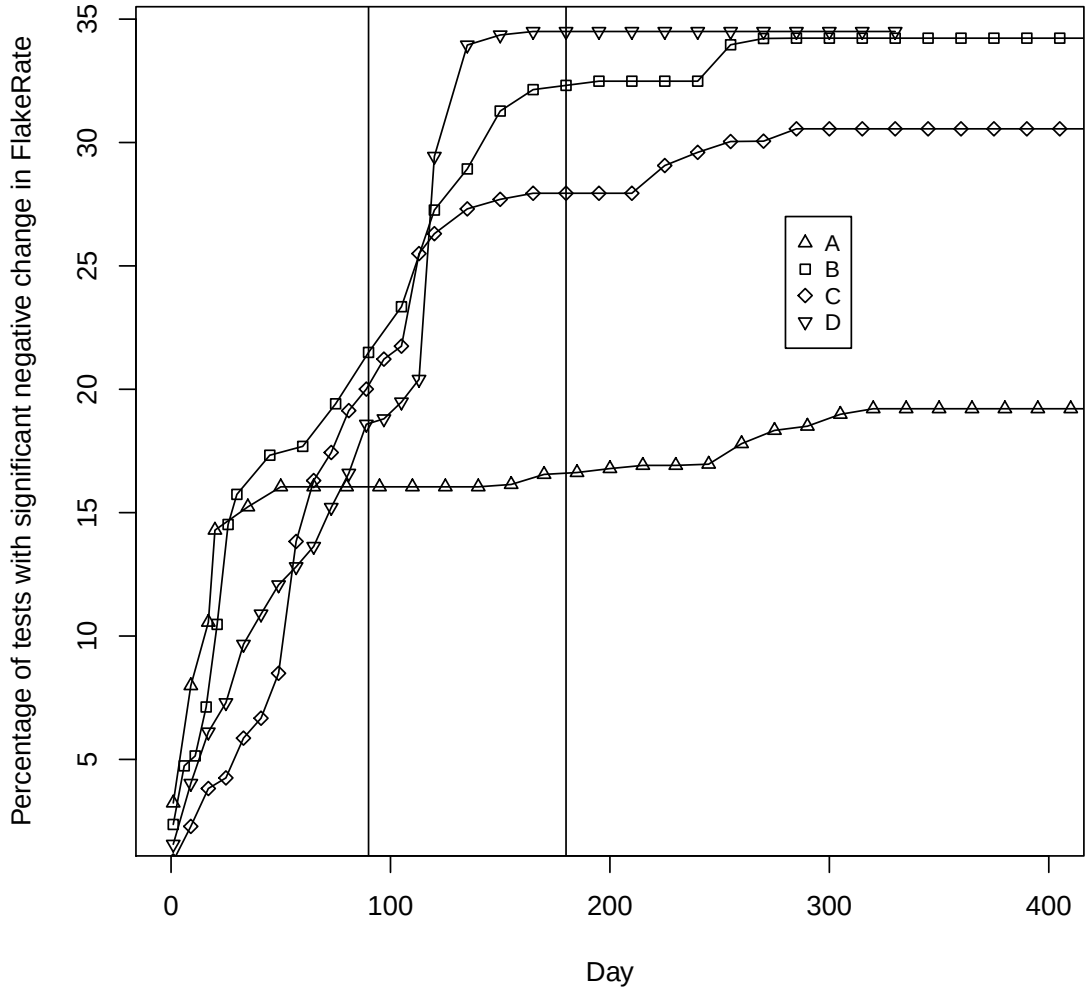

Figure 6: RQ3: The cumulative proportion of tests that deviate from the StableFlakeRate at least once the release cycle.

4.4 RQ4: Savings

How many test failures are likely flaky failures that can be ignored?

Without quantifying test flakiness or a mechanism to determine test stability, testers must investigate all tests that fail. However, we argue that flaky tests only need to be investigated when they deviate from their StableFlakeRate, *i.e.* the flake rate of the last stable release. In this section, we examine how many test failures can be ignored because there is no statistically significant change in the FlakeRate.

Figure 7 shows the cumulative percentage of tests that fail at least once over each release cycle. Before the midpoint of the release, day 90, we observe that the number of tests that fail increases rapidly. After day 90 Release D sees a sharp increase in test fails from 52% to 68% at day 119. After day 180 when the release is rolled out the total number of failed tests in Release D plateaus. Release A, B, and C see a slight increase in the percentage of tests that fail after day 180.

The number of tests that fail in a release, Figure 7, is substantially higher than the number of tests that become unstable for the same release, see Figure 6 in the previous section. As a result, substantial investigation effort can be saved by ignoring failures that do not statically deviate from their StableFlakeRate. To illustrate the potential savings in test investigation, we subtract percentage of tests that are unstable from the total failed tests. Figure 8 shows the savings as percentage of failed tests that can be ignored for each release. We see that halfway through a release at day 90, testers will have to examine 35%, 36.5%, 29%, and 30.5% fewer tests for release A, B, C and D, respectively. At the time when the release is rolled out at day 180 testers will examine 40.5%, 42%, 33.7% and 39% fewer tests for release A, B, C and D respectively.

The savings for release A, C, and D are similar at day 100. The rise in saving in rapid form day 1 to day 90 for each release as compared to the savings in the second half of the release *i.e.* form day 90 to 180. In the second half of the release cycle the savings increase by 5%, 5.5%, 6% and 9.5% for release A, B, C, and D, respectively.

Instead of ignoring the flaky fails testers can prioritize their investigation of failed tests. Unstable tests are more likely due to a real fault in the system. After investigating the tests that are unstable testers can investigate "normal" failures to look for faults and potential environmental problems.

> Compared to investigating all failed tests, if testers only investigate statistical significant deviations from the StableFlakeRate, then we would investigate 35% to 42% fewer tests

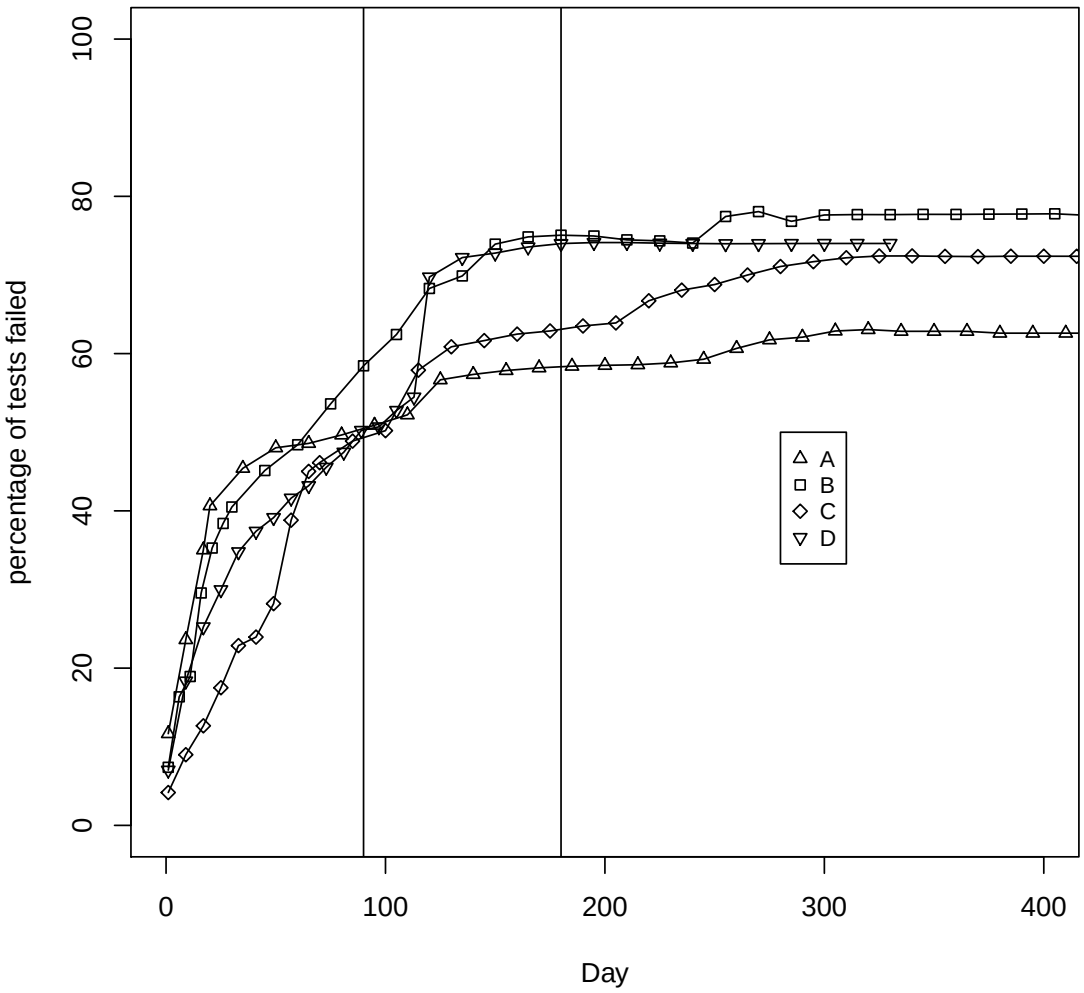

Figure 7: RQ4: The percentage of tests that have failed at least one time over the release cycle.

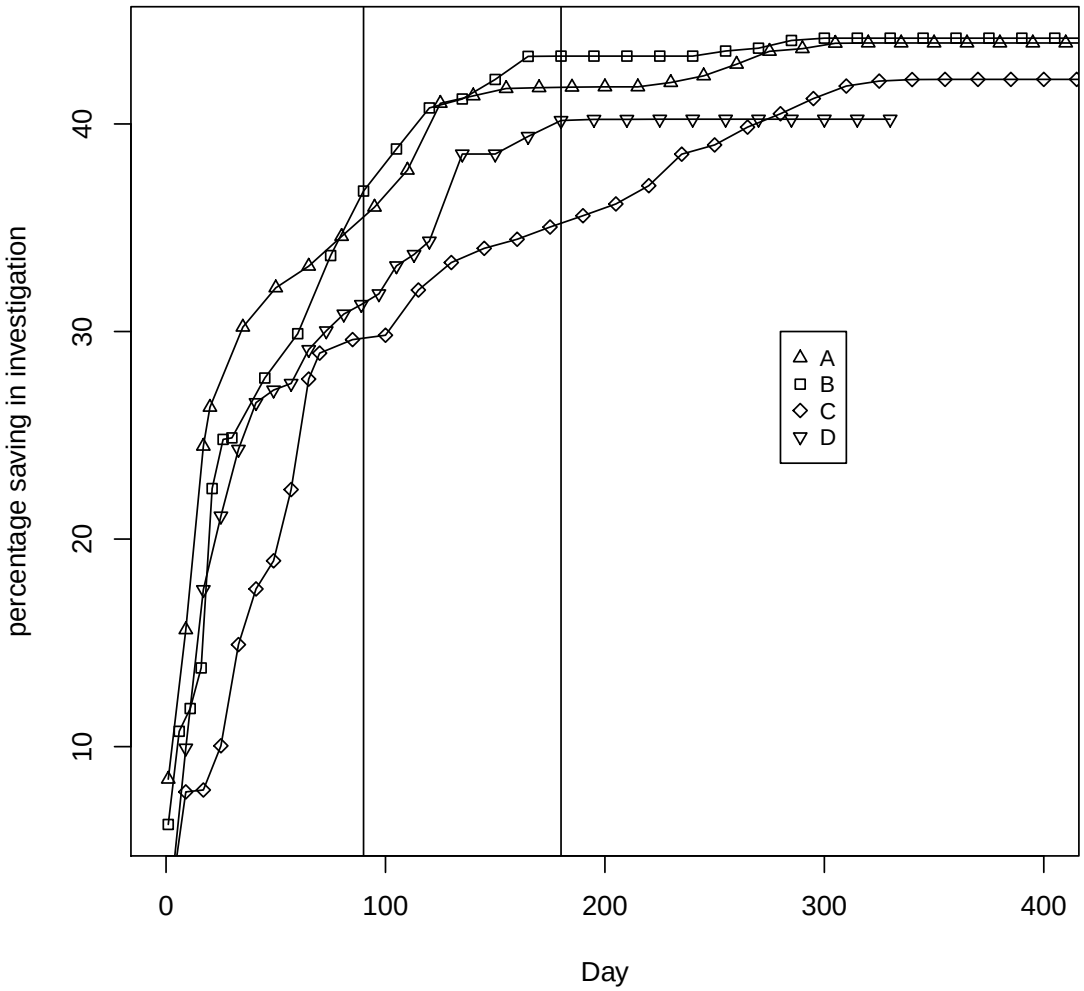

Figure 8: RQ4: Savings in investigation effort. The difference of total failed tests minus those that are unstable.

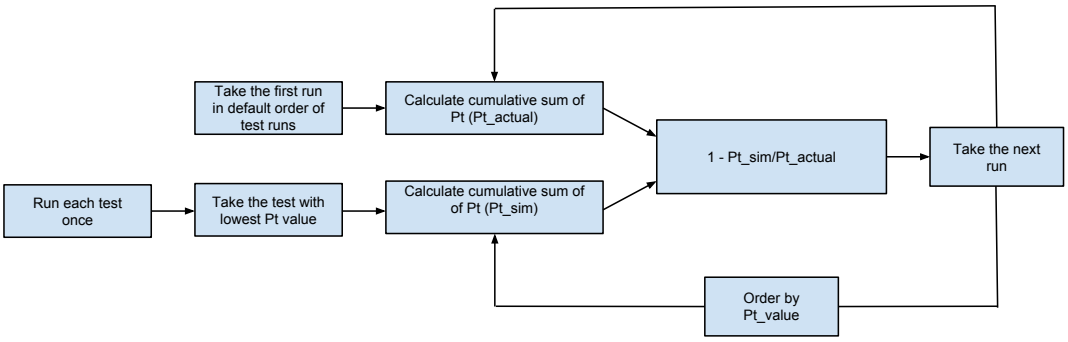

Figure 9: RQ5: BinomialStabilityOrder algorithm simulation methodology

4.5 RQ5: Prioritizing Reruns

Can we prioritize test runs to find test instabilities sooner?

Flaky tests require multiple runs to determine if there has been a change in the StableFlakeRate which warrants investigation. Our goal is to look for tests with unlikely result sets and run them ahead of tests with likely result sets. We use the binomial distribution to determine for a test, t, the probability, P_t, of a result set with f failures given r runs and a StableFlakeRate:

$P_t(f, r, \text{StableFlakeRate}) =$

$$\binom{r}{f} * \text{StableFlakeRate}^f * (1 - \text{StableFlakeRate})^{r-f} \qquad \text{(restatement of Eq. 9)}$$

Our simulation methodology is illustrated in Figure 9. In the figure P_t_actual is the cumulative sum of the P_t values of tests that have been run in the default order. P_t_sim is the cumulative sum of P_t values of tests that have been run after ordering test runs in ascending order of P_t.

To create a DefaultOrder baseline, for each release, we order each test run by the date and time it was run. We then calculate P_t. In contrast, for the BinomialStabilityOrder algorithm, we prioritize runs in ascending order of P_t to ensure that the most unstable tests are re-run ahead of stable tests. We first run each test once, to determine the initial P_t value. We queue the tests from lowest to highest P_t value. We select the test with the lowest P_t value to be run and then insert it back into the sorted queue based on its new P_t value.

We define the change in test stability for a release as the cumulative difference of percentage decrease in P_t. We define the ReleaseTestStability for a release as the cumulative difference in P_t values between the BinomialStabilityOrder and DefaultOrder given a set of r runs:

$$\textsc{ReleaseTestStability}(Release) = 1 - \frac{\sum_{n=1}^{r} P_{\textsc{BinomialStabilityOrder}}}{\sum_{n=1}^{r} P_{\textsc{DefaultOrder}}} \tag{10}$$

The BinomialStabilityOrder algorithm requires each test to be run once, so the total number of runs we have available to re-prioritize is

$$R = TotalAvailableReruns = TotalTestRuns - NumberOfTests \tag{11}$$

For simplicity, we define $r = 1$ to be the first test run after each test has been run once. Our simulation is performed on historical test results, so once we have used all of the runs for a test in our simulation, we cannot use more reruns. As a result, even though a test may still have the lowest P_t we cannot re-run it. As a colliery, when $r = R = TotalAvailableReruns$ then $\sum_{n=1}^{R} P_{\textsc{BinomialStabilityOrder}} = \sum_{n=1}^{R} P_{\textsc{DefaultOrder}}$.

Figure 10, plots the difference in ReleaseTestStability for each release against the percentage of total test runs. We see that the first runs are allocated to tests that have failed for the first time and have a $P_t = 0$, so the decrease in ReleaseTestStability is 100%. After 15% of the total runs for a release, we see a 30% to 78% decrease in the ReleaseTestStability for that release. After 30% of total runs have been used we see a 9% to 38% decrease in ReleaseTestStability. After 50% of runs the decrease in ReleaseTestStability for Release A, B, and D is less then 10%, whereas for Release C the decrease in ReleaseTestStability is 19%. When 100% of the runs have been used, there is no difference between the BinomialStabilityOrder and the DefaultOrder.

We develop the BinomialStabilityOrder algorithm to order test based on their instability. Using only 15% of the total runs for a release, we see a decrease of 30% to 78% in ReleaseTestStability. With fewer test runs, testers can find the unstable test and investigate them earlier in the release cycle.

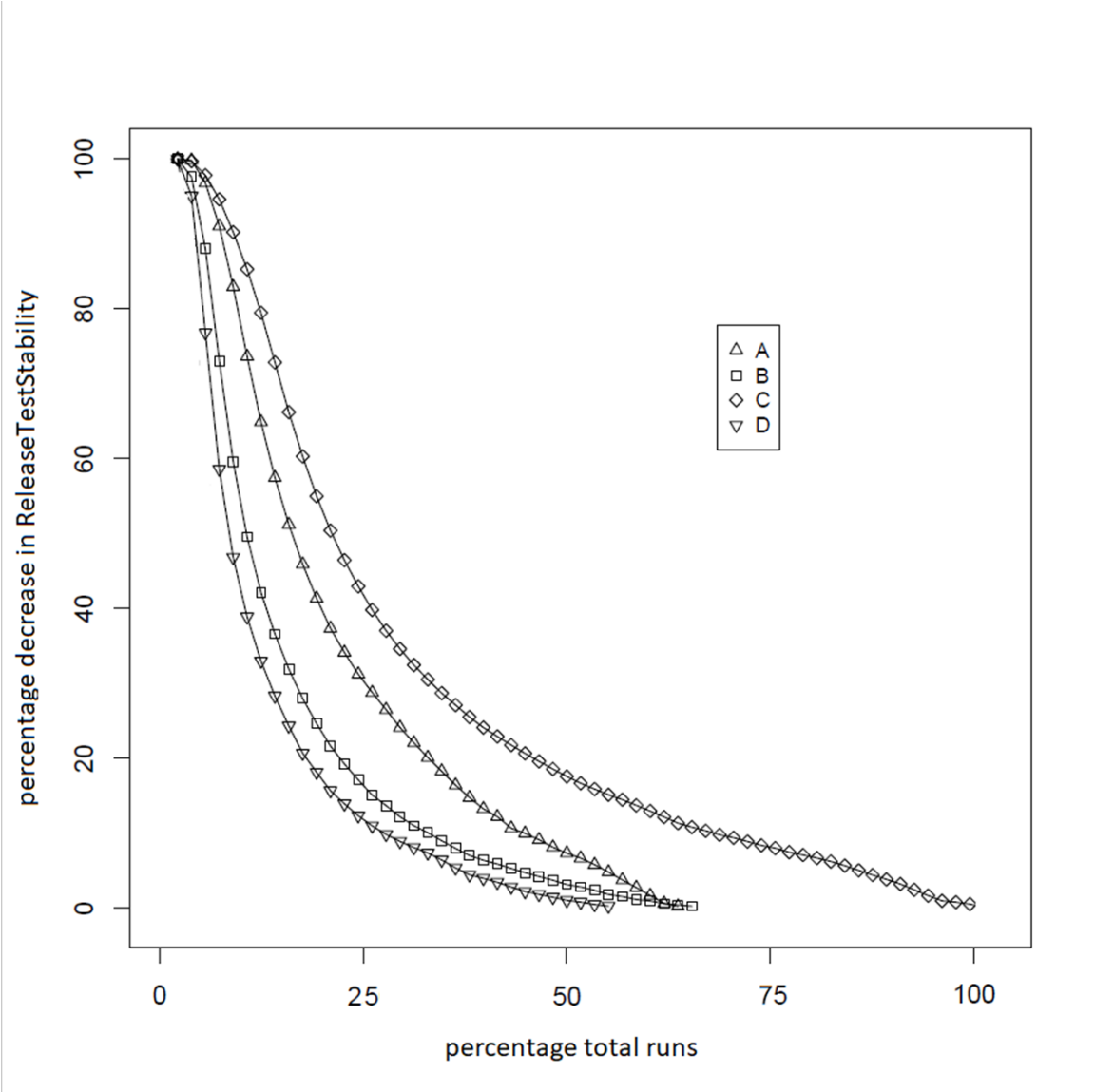

Figure 10: RQ5: Prioritizing re-runs. The difference in ReleaseTestStability between the DefaultOrder with the BinomialStabilityOrder. The BinomialStabilityOrder algorithm finds test instability in fewer runs.

4.6 Threats to Validity

There is a threat to generalizability because this study was conducted on a single project at Ericsson. However, the definitions and measures are not dependent on this particular project and it should be easy to replicate the study on other projects.

We were unable to re-run tests and relied on the historical test outcomes. As a result, we were unable to re-run the same test on a stable build to determine its StableFlakeRate as is done at Microsoft [39] and Facebook [53]. The system under test or even the test itself may have changed which may impact the FlakeRate. However, our goal was to find test instabilities, so if a test or the underlying code has been re-written the BinomialStabilityOrder algorithm will quickly flag it for investigation and its StableFlakeRate can be reset.

Tests report more failures in the initial stage of testing because there are product faults in the system under test which leads to real fails. Some of the real fails i.e. fails due to product fault can be misidentified as flaky of the there is no bug report attached with it. The real and the flaky fails are intertwined in the data significantly in the first half of testing. Due to this reason we use only the test runs in the stable period of the release in calculating the StableFlakeRate so that the probability of a test failing due to product fault and misidentified as a flaky fail is very small. This problem is inherent in the data which can raise some alarms about the validity of our approach. This workaround of using the stable period was adopted after discussion with the test manager.

After identifying the tests that become unstable we do not identify if these tests raise bug reports. Future work is necessary to determine if test instability is linked to product faults.

Chapter 5

Tool Deployment and other implications for Ericsson

Our findings were met with resistance at Ericsson. Test runs at Ericsson are expensive because they require specialized base-station software. Informing developers and testers that their tests are highly flaky and must be run tens to hundreds of times to provide even 95% confidence in the result is impractical. There were two tangible outcomes to this work. First, testers requested an order list of the most flaky tests by FlakeRate. This list was used to examine some of the more flaky tests and resulted in some test being fixed or removed. Other flaky tests were moved to earlier testing phases that run in a controlled virtual environment instead on actual base-station hardware. Second, testers were overwhelmed by the number of test failures they had to investigate in a single day. We developed a preliminary tool to flag and rank when a test is statistically unstable.

5.1 The FlakeFinder tool

Developing an appropriate measure of test flakiness in the FlakeRate required multiple iterations. In early work, we implemented a tool that flagged tests for which the failure rate was outside a 95% confidence interval of sequence of previous failure rate. This idea, was implemented as a tool that Ericsson developers used on a daily basis in the scrum meetings. The tool divided the failed tests into two categories. If the current failure rate was greater than the 95% confidence interval of the historical trend of failures the test fail was included in the list of 'potential product faults.' If the current failure rate of tests was less then 95% confidence interval of historical fail rate than the test was included in the list of 'potential flaky fails'.

5.1.1 Statistical Anomaly Detection

In order to implement the statistical anomaly detection algorithm, Ericsson testers requested that we use a time frame of size 30 days of test runs.[1] For all tests, we record the total daily runs, and the total daily failures for each day over a 30 day time frame. We than divide the number of times a test failed for each day by total runs on that day to normalize. For each test case the result is a sequence of length 30 containing the normalized fails. The vector is ordered is ascending order of normalized fails. For every test such a vector was obtained. We call this vector the baseline non-determinism vector.

..... 0.1, 0.3, 0.42, 0.5, 0.61, 0.69, 0.75, 0.8, 0.82, 0.9, 0.95, ...

The next step is to calculate the 95% percent confidence interval of this vector. We use the quantile function in R. If the normalized number of fails for the current day is greater then the 95th percentile, we flag the fail as an anomaly and should be investigated. For example, if a test is run 10 times and it fails 7 out of 10 times, the normalized number of fails are 0.7. If the 95% confidence interval is 0.67 then the test fail is an anomaly and we flag it for investigation.

5.1.2 Magnitude of Anomaly

We calculate the magnitude of anomaly as the absolute value of difference between the current normalized failure rate and the 95% confidence interval. We subtract the value at the 95th percentile from the current normalized fail rate. If the difference is greater then 0 then it means that the test fail is an anomaly. The absolute difference gives the degree of anomaly. For example if current failure rate of a tests is 0.47 and the 95% confidence interval value is 0.25, the magnitude of anomaly is 0.22. We order the failed tests by the magnitude of anomaly. The idea is that greater the magnitude of anomaly the greater is the potential that the test failed due to a product fault. The steps are illustrated in Figure 11.

5.1.3 FlakeFinder in action

Figure 12 shows a snapshot of FlakeFinder tool. The figure contains the list of test fails for a particular day. The first columns is the 'Rerun passes in last month' that indicates if the failed test passed on rerun within the last month (30 days of testing). The test field is the name of the test case. The date field shows the date of the execution of the test. TGF Job(s) is the job id which was used to execute the tests. The STP is the id of hardware component the was used to execute the test. Testers have the option to filter the results based on STP in case the list is very long. The upgrade package is the id of the update patch under test.

[1] We used a range of time frames including the entire release period and the previous release

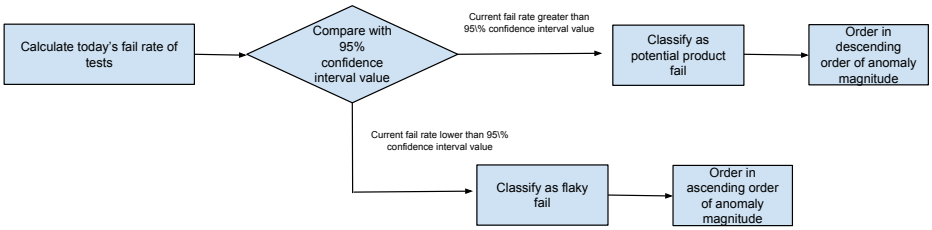

Figure 11: The FlakeFinder tool steps

STP Filter Go

Rerun Passes in last month	Test	Date	TGF Job(s)	STP	Upgrade Package	Investigation Priority	TR Identified?
0	1_scheduling_PFS_SUITE:tc_sa_max_c_over_i_dl - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	100%	☑
0	1_scheduling_PFS_SUITE:tc_sa_max_c_over_i_ul - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	100%	☐
0	1_scheduling_PFS_SUITE:tc_sa_pf_low_dl - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	100%	☐
0	1_scheduling_PFS_SUITE:tc_sa_pf_low_ul - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	100%	☐
0	1_scheduling_PFS_SUITE:tc_sa_pf_medium_ul - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	94%	☐
0	1_scheduling_PFS_SUITE:tc_sa_resource_fair_ul - scheduling_tests	2017-07-13	11803967	OTENB1010	CXP102051/27_R20A07	94%	☐

Figure 12: Snapshot of the FlakeFinder tool in action

The test fails are ordered in descending order of investigation priority. The investigation priority is based on the magnitude of anomaly discussed above. A test with highest value of magnitude of anomaly is ranked at the top of the list The last column 'TR-identified' is used to collect feedback from the testers in real time to measure the effectiveness of the ranking. In case there was a Trouble report raised for the failed tests the testers are expected to check the Tr identified field. This enabled us to measure the effectiveness of the anomaly detection technique without mining the data from archived history of test runs.

5.1.4 Tool Evaluation

We evaluated how effective the anomaly detection technique was in identifying test fails due to product faults and also how much investigation effort can be saved by ignoring the test fails for which the failure rate was lower then 95% confidence interval. We ran this simulation on the historical data. The preliminary results showed that by dividing the failed tests into two separate lists with one containing the test fails potentially due to product faults and the other containing potential flaky fails that can be ignored testers could detect 31% true alarms with a savings in investigation effort of 74%.

A testing team at Ericsson used the tool each morning to prioritize the failing tests from the nightly run. Initially, the tool proved popular, however, the the use of confidence intervals was not sufficiently sophisticated to provide enough accuracy to be trusted by developers. As the more sophisticated binomial based analysis shows, there is more runs and more flaky tests than can be handled in a single nightly run of testing.

Chapter 6

Conclusion and Future Work

Software testing is increasing in complexity as software becomes more ubiquitous and critical to the daily functioning of life. With an increase in complexity comes a need to make the testing process more transparent and streamlined. Flaky tests are a big hurdle in achieving this objective because their test outcomes are not reliable. Flakiness effect a large subset of test cases at Ericsson (74%), so quarantining flaky tests is not a viable option. In addition Google and Facebook also report that flaky tests reduce release speed and waster engineer's time. In this thesis, we quantify flaky tests with the FlakeRate, determine how confident developers can be in their test outcomes during a release, quantify test stability relative to the previous release, and suggest an algorithm to re-run unstable tests to find faults sooner in the release cycle. We make five specific contributions.

1. Companies treat tests as either flaky or not [53, 39**?**]. In contrast, we quantify FlakeRate of each test. We further differentiate between the rate of the last stable release StableFlakeRate and the current FlakeRate. Studying Ericsson, we find that between 24% to 36% have never had a flaky failure. Despite the environmental noise at Ericsson between 56% to 76% of tests have flaked less then 5/100 times and only 18% to 22% of tests have flaked more then 10/100 times.

2. At Google, a test labeled as flaky must fail three times in a row to be investigated [**?**]. In contrast to this arbitrary value, we use the standard statistical sample size formula to determine how confident we can be in test given the number of times at has been run. At Ericsson, we find that at the midpoint of the release cycle, 68% to 80% of test have been run a sufficient number of times to be 95% confident, at the end of the release the corresponding percentages are 78% to 83% of tests.

3. Flaky tests have inherent noise leading them to fail when there is no fault. We use the

the binomial test, to find any statistically significant deviation of the FlakeRate from the expected StableFlakeRate distribution. These deviation indicate test instability and should be investigated as a potential product fault by developers. At Ericsson, at the end of a release 17% to 34% of total tests will have been flagged as unstable and should be investigated by a tester.

4. A failing test should be investigated, however, flaky tests reduce tester confidence in test outcomes. Tests that exhibit instability can be investigated before other failing tests because unstable tests are more likely to indicate a faulty change in the software system. At Ericsson, if testers only unstable tests that have a statistical significant deviations from the StableFlakeRate, then they would save 35% to 42% of failed test investigations.

5. Flaky tests require multiple runs to statistically determine if they vary from the StableFlakeRate and are unstable. We develop the BinomialStabilityOrder algorithm to order test based on their instability. At Ericsson, re-prioritizing tests runs to re-run the least stable test first, we see that with only 15% of the total runs for a release, there is a decrease in test stability of 30% to 78%. With fewer test runs, developers can find the unstable test and investigate them earlier in the release cycle.

   There is a stigma around flaky tests and much future work. Developing an appropriate measure of test flakiness in the FlakeRate required multiple iterations and many failed attempts. Quantifying instability using the binomial distribution is obvious in retrospect, but as indicated by our "confidence interval" tool was not obvious at the time and is still difficult to explain to testers. We describe four specific areas of future work. First, displaying changes in the FlakeRate and prioritizing failing test reruns in a tool will require willingness from a development and much prototyping. Second, project managers want to see flaky tests fixed and allowing flaky tests to remain in the system so long as they are unstable may lead to testers ignoring flaky tests. One possible solution to this problem comes in that our BinomialStabilityOrder algorithm runs failures of tests with low FlakeRates before tests with high FlakeRates because a failure of a test with a low FlakeRate is less likely. As a result, to have a flaky test re-run, the developers should fix or at least reduce the FlakeRate of their tests. Third, the results presented here are raw Ericsson FlakeRates and are unlikely to be published due to a non-disclosure agreement. Instead, we plan to simulate these results using a variety of published flake rates, including those from Google [**?**] and Facebook [53]. Finally, we hope to change the attitude of engineers who assume that all flaky tests are equally bad. We show that flaky tests provide value and can quantified through the FlakeRate and used to find tests instability that is indicative of a potential product fault.

Bibliography

[1] Android flakytests annotation. `http://goo.gl/e8PILv`.

[2] Jenkins randomfail annotation. `http://goo.gl/tzyC0W`.

[3] Severity rating. `https://goo.gl/ije3Qk`.

[4] Spring repeat annotation. `http://goo.gl/vnfU3Y`.

[5] A. Agresti and B. A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[6] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 312–321. IEEE, 2013.

[7] B. Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] B. Beizer. Software testing techniques, van nostrand reinhold. *Inc, New York NY, 2nd edition. ISBN 0-442-20672-0*, 1990.

[9] J. Bell and G. Kaiser. Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, New York, NY, USA, 2014. ACM.

[10] J. Bell, O. Legunsen, M. Hilton, Eloussi, Yung, and Marinov. Efficiently and automatically detecting flaky tests with deflaker. `https://goo.gl/w5jSKg`, 2018.

[11] J. Bell, O. Legunsen, M. Hilton, Eloussi, Yung, and Marinov. Get rid of your flakes. `https://goo.gl/aCrRuk`, 2018.

[12] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: Automatically detecting flaky tests. 2018.

[13] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, Apr. 2001.

[14] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[15] L. D. Brown, T. T. Cai, and A. DasGupta. Confidence intervals for a binomial proportion and asymptotic expansions. *Ann. Statist.*, 30(1):160–201, 02 2002.

[16] B. Busjaeger and T. Xie. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 975–980. ACM, 2016.

[17] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[18] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[19] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 155–160, New York, NY, USA, 2010. ACM.

[20] C. Catal and D. Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[21] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[22] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, pages 211–220. IEEE Computer Society Press, 1994.

[23] W. Cochran. Sampling techniques, new york, 1953. *Statistical Surveys E. Grebenik and CA Moser*, 1963.

[24] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 357–366. IEEE, 2011.

[25] N. Dini, A. Sullivan, M. Gligoric, and G. Rothermel. The effect of test suite type on regression test selection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 47–58. IEEE, 2016.

[26] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 71–82. ACM, 2008.

[27] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 411–420. IEEE, 2005.

[28] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[29] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[30] S. Elbaum, G. Rothermel, and J. Penix. 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, pages 235–245, New York, NY, USA, 2014. ACM.

[31] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[32] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[33] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[34] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[35] HaJaeheon, H. Dinges, Manson, and Meng. System to uncover root cause of non-deterministic (flaky) tests. `https://goo.gl/U635cs`, 2016.

[36] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 291–301, May 2002.

[37] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[38] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[39] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 39–48. IEEE Press, 2015.

[40] G. D. Israel. *Determining sample size*. University of Florida Cooperative Extension Service, Institute of Food and Agriculture Sciences, EDIS Gainesville, 1992.

[41] J. Kasurinen, O. Taipale, and K. Smolander. Test case selection and prioritization: risk-based or design-based? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10. ACM, 2010.

[42] J.-M. Kim and A. Porter. 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*, pages 119–129, May 2002.

[43] L. Kish. Survey sampling. 1965.

[44] A. Labuschagne, L. Inozemtseva, and R. Holmes. 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*, ESEC/FSE 2017, pages 821–830, New York, NY, USA, 2017. ACM.

[45] E. Lamyaa. Determining flaky tests from test failures. Master's thesis, University of Illinois, Urbana-Champaign, 2015.

[46] H. K. Leung and L. White. Insights into regression testing (software testing). In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69. IEEE, 1989.

[47] H. K. Leung and L. White. A study of integration testing and software regression at the integration level. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 290–301. IEEE, 1990.

[48] H. K. N. Leung and L. White. Insights into regression testing [software testing]. In *Proceedings. Conference on Software Maintenance - 1989*, pages 60–69, Oct 1989.

[49] H. K. N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings. Conference on Software Maintenance 1991*, pages 201–208, Oct 1991.

[50] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698. ACM, 2018.

[51] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.

[52] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. Assessing test case prioritization on real faults and mutants. *arXiv preprint arXiv:1807.08823*, 2018.

[53] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 91–100. IEEE Press, 2019.

[54] M. N. Marshall. Sampling for qualitative research. *Family Practice*, 13(6):522–526, 1996.

[55] S. McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2nd edition, June 2004.

[56] A. M. Memon and M. B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.

[57] S. T. C. S. N. Meng. An empirical study of flaky tests in android apps.

[58] G. Miaoulis and R. D. Michener. *An introduction to sampling*. Kendall, 1976.

[59] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering*, pages 222–232. ACM, 2018.

[60] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen. On rapid releases and software testing. In *2013 IEEE International Conference on Software Maintenance*, pages 20–29, Sept 2013.

[61] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 496–499, New York, NY, USA, 2011. ACM.

[62] W. Piegorsch. Sample sizes for improved binomial confidence intervals. *Computational Statistics and Data Analysis*, 46(2):309–316, 6 2004.

[63] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86. ACM, 2008.

[64] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 130–140, New York, NY, USA, 2002. ACM.

[65] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug 1996.

[66] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[67] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[68] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43, Nov 1998.

[69] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *icsm*, page 34. IEEE, 1998.

[70] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[71] P. R. Srivastava. Test case prioritization. *Journal of Theoretical & Applied Information Technology*, 4(3), 2008.

[72] S. Sudman. Applied sampling. Technical report, Academic Press New York, 1976.

[73] J. Thomas. Testing at the speed and scale of google. `https://goo.gl/kEdQea`, 2011.

[74] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 101–110. IEEE, 2015.

[75] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen. Multi-objective test prioritization in software product line testing: an industrial case study. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 32–41. ACM, 2014.

[76] S. Wang, J. Nam, and L. Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 523–534. ACM, 2017.

[77] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[78] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[79] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

[80] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*, pages 522–528. IEEE, 1997.

[81] T. Yamane. Statistics, an introductory analysis 2nd edition: Horper and row. *New York*, 1967.

[82] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[83] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 385–396, New York, NY, USA, 2014. ACM.

[84] Y. Zhang, X. Yang, and H. Li. An outlier mining algorithm based on confidence interval. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 231–234, April 2010.