Data Science for Software Maintenance

by

Laura Inozemtseva

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

© Laura Inozemtseva 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Denys Poshyvanyk Associate Professor Computer Science Department The College of William and Mary
Supervisors	Reid Holmes Associate Professor Department of Computer Science University of British Columbia
	Michael Godfrey Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Members	Joanne Atlee Professor David R. Cheriton School of Computer Science University of Waterloo
	Meiyappan Nagappan Assistant Professor David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member	Derek Rayside Assistant Professor Electrical and Computer Engineering University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation:

- Reid Holmes
- Adriaan Labuschagne
- René Just
- Darioush Jalali
- Michael Ernst
- Gordon Fraser

Abstract

Maintaining and evolving modern software systems is a difficult task: their scope and complexity mean that seemingly inconsequential changes can have far-reaching consequences. Most software development companies attempt to reduce the number of faults introduced by adopting maintenance processes. These processes can be developed in various ways. In this thesis, we argue that data science techniques can be used to support process development. Specifically, we claim that robust development processes are necessary to minimize the number of faults introduced when evolving complex software systems. These processes should be based on empirical research findings. Data science techniques allow software engineering researchers to develop research insights that may be difficult or impossible to obtain with other research methodologies. These research insights support the creation of development processes. Thus, data science techniques support the creation of empirically-based development processes.

We support this argument with three examples. First, we present insights into automated malicious Android application (app) detection. Many of the prior studies done on this topic used small corpora that may provide insufficient variety to create a robust app classifier. Currently, no empirically established guidelines for corpus size exist, meaning that previous studies have used anywhere from tens of apps to hundreds of thousands of apps to draw their conclusions. This variability makes it difficult to judge if the findings of any one study generalize. We attempted to establish such guidelines and found that 1,000 apps may be sufficient for studies that are concerned with what the majority of apps do, while more than a million apps may be required in studies that want to identify outliers. Moreover, many prior studies of malicious app detection used outdated malware corpora in their experiments that, combined with the rapid evolution of the Android API, may have influenced the accuracy of the studies. We investigated this problem by studying 1.3 million apps and showed that the evolution of the API does affect classifier accuracy, but not in the way we originally predicted. We also used our API usage data to identify the most infrequently used API methods. The use of data science techniques allowed us to study an order of magnitude more apps than previous work in the area; additionally, our insights into infrequently used methods illustrate how data science can be used to guide API deprecation.

Second, we present insights into the costs and benefits of regression testing. Regression test suites grow over time, and while a comprehensive suite can detect faults that are introduced into the system, such a suite can be expensive to write, maintain, and execute. These costs may or may not be justified, depending on the number and severity of faults the suite can detect. By studying 61 projects that use Travis CI, a continuous integration system, we were able to characterize the cost/benefit tradeoff of their test suites. For

example, we found that only 74% of non-flaky test failures are caused by defects in the system under test; the other 26% were caused by incorrect or obsolete tests and thus represent a maintenance cost rather than a benefit of the suite. Data about the costs and benefits of testing can help system maintainers understand whether their test suite is a good investment, shaping their subsequent maintenance decisions. The use of data science techniques allowed us to study a large number of projects, increasing the external generalizability of the study and making the insights gained more useful.

Third, we present insights into the use of *mutants* to replace real faulty programs in testing research. Mutants are programs that contain deliberately injected faults, where the faults are generated by applying *mutation operators*. Applying an operator means making a small change to the program source code, such as replacing a constant with another constant. The use of mutants is appealing because large numbers of mutants can be automatically generated and used when known faults are unavailable or insufficient in number. However, prior to this work, there was little experimental evidence to support the use of mutants as a replacement for real faults. We studied this problem and found that, in general, mutants are an adequate substitute for faults when conducting testing research. That is, a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed, for both developer-written and automatically-generated test suites. However, we also found that additional mutation operators should be developed and some classes of faults cannot be generated via mutation. The use of data science techniques was an essential part of generating the set of real faults used in the study.

Taken together, the results of these three studies provide evidence that data science techniques allow software engineering researchers to develop insights that are difficult or impossible to obtain using other research methodologies.

Acknowledgements

I'd like to thank my supervisor, Reid Holmes, who had a tremendous impact on me throughout my seven years at Waterloo. Though grad school is ostensibly about sharpening one's technical skills, the most valuable things I learned from Reid were about the social aspects of research: how to present my work well, how to communicate its value, and how to win friends and influence people at conferences. Reid also gave me an inside look at the life of a professor by involving me in activities grad students are often not entrusted with, such as grant writing and course design. These experiences proved invaluable when deciding on my career path. Reid was also very flexible with my unusual (to put it mildly) decision to move to California and have children in the middle of my PhD. I could not possibly have juggled family and school without his support. All in all, I thoroughly enjoyed working together and couldn't have asked for a better supervisor (even if I acquired a habit of abusing parentheses).

I'd also like to thank Mike Godfrey for piquing my interest in software engineering with his grad course about software evolution and for introducing me to the SWAG family. When Reid was on vacation prior to my defence, Mike stepped in and provided a tremendous amount of editorial assistance. My thesis and my presentation were greatly improved by his comments. In addition, his wife Anita and son Trevor generously babysat for me while I defended my thesis.

Thanks also go to my former supervisor, Therese Biedl, who took a chance on me by accepting me to Waterloo. Though I didn't end up staying in the algorithms group, I would not have been in grad school at all without her.

Of course, my other committee members were also a vital part of completing this work. Thanks to Jo Atlee, Derek Rayside, Mei Nagappan, and Denys Poshyvanyk for their time and their comments.

I'd also like to thank my family. My parents Robert Bogdan and Janet and Paul Ennis have provided a tremendous amount of support in both emotional and material ways. My inlaws, Svetlana Inozemtseva and Mikhail Inozemtsev, have also been very helpful. In particular, I would not have been able to complete my internship at Microsoft Research without both grandmas generously staying with us for six weeks each to provide childcare. I owe more to all five of them than I could list here.

Finally, I'd like to thank my husband, Greg Inozemtsev, without whose support this thesis would never have been possible. He was always happy to discuss my research when I needed another point of view and prevented me from rage-quitting on more than one occasion. His crazy plans, like moving to California, add lots of excitement to my life in

both good and bad ways. (Mostly good.) I'm very happy to have someone like him to spend my life with; I'm glad I made a spontaneous decision to join the Queen's robotics team back in 2006. Our sons Fyodor and Anton Inozemtsev have also made my life much richer even if they made it harder to find time to do research. Though they're too young to fully understand the concept of grad school, I can still teach them to call me Dr. Mom.

Dedication

For my mom, Janet Munro Ennis, whose intelligence, generosity, and kindness are matched only by her courage in her struggle with cancer. I love you, Mom.

Table of Contents

List of Tables xix

List of Figures xxi

1 Introduction 1

1.1 Roles and Publication Status 5

2 Detecting Android Malware 7

2.1 Related Work 9

2.1.1 Corpus Size Guidelines 9

2.1.2 Malware Detection 11

2.2 RQ1: How Many Applications? 14

2.2.1 Method 14

2.2.2 Results 20

2.3 RQ2: Should API Level Be Controlled? 26

2.3.1 Method 26

2.3.2 Results 35

2.4 Discussion 42

2.5 Threats to Validity 45

2.6 Relevance to Thesis 47

3 **Measuring Test Suite Costs and Benefits** **49**
3.1 Related Work 51
3.1.1 Measuring Test Suite Maintenance Effort 52
3.1.2 Capturing Test Outputs 53
3.1.3 Reducing Suite Execution Time 53
3.2 Method 55
3.3 Collecting Travis Build Data 55
3.3.1 Identifying Build Status 61
3.3.2 Identifying Build Transitions 61
3.3.3 Mapping Builds to Code Changes 63
3.3.4 Classifying Failure Resolutions 64
3.4 Assessing the Costs and Benefits of Regression Testing 64
3.4.1 RQ1: What Fraction of Tests Are Flaky? 64
3.4.2 RQ2: How Often Are Failures Beneficial? 66
3.4.3 RQ3: Why Do Tests Usually Require Maintenance? 68
3.5 Implications for Test Selection 72
3.5.1 RQ4: How Often Do Tests Expose Faults? 72
3.6 Threats to Validity 74
3.7 Replication Package 75
3.8 Relevance to Thesis 75

4 **The Use of Mutants in Testing Research** **77**
4.1 Method 80
4.1.1 Subject Programs 80
4.1.2 Locating and Isolating Real Faults 81
4.1.3 Obtaining Developer-written Test Suites 83
4.1.4 Automatically Generating Test Suites 86
4.1.5 Mutation Analysis 87

4.1.6 Experiments 88
4.2 Results 91
4.2.1 Are Real Faults Coupled to Mutants Generated by Commonly Used Mutation Operators? 91
4.2.2 What Types of Real Faults Are Not Coupled to Mutants? 93
4.2.3 Is Mutant Detection Correlated with Real Fault Detection? 97
4.2.4 Threats to Validity 99
4.3 Related Work 102
4.3.1 Studies That Explored the Relationship Between Mutants and Real Faults 102
4.3.2 Commonly Used Artifacts 103
4.3.3 Software Testing Research Using Mutants 104
4.4 Relevance to Thesis 104

5 Conclusion **107**

References **111**

List of Tables

2.1 The number of Android methods introduced and removed in each API level (version) and the total number of methods in each level. 17

2.2 P values of the Kolmogorov-Smirnov (K-S) and Kuiper tests comparing each smaller corpus to the largest corpus (1,000,000 apps). The null hypothesis is that the data sets are drawn from the same population distribution function. The alternative hypothesis is that the two experimental distributions are consistent with the same theoretical distribution. The second and third columns give P values for the distributions of distinct methods used, while the fourth and fifth columns give P values for the distributions of distinct levels used. 24

2.3 The percentage of the most popular 20 API methods in each corpus that match the most popular 20 API methods in the largest corpus, both when order is considered and when it is ignored. 24

2.4 The percentage of the least popular 20 API methods in each corpus that match the least popular 20 API methods in the largest corpus, both when order is considered and when it is ignored, including methods with zero calls. 25

2.5 The percentage of the least popular 20 API methods in each corpus that match the least popular 20 API methods in the largest corpus, both when order is considered and when it is ignored, excluding methods with zero calls. 25

2.6 The number of apps from our dataset assigned to each level. The third and fourth columns give the number of malicious apps as a count and as a percentage of the total, respectively. The table includes the 1,295,091 apps from Google Play and the 7,036 from other sources. 33

2.7 The ten least popular methods from the Android API in our dataset. None of the methods were called by apps in our dataset. 46

3.1 The 61 projects considered in this study. The second column lists the number of non-blank, non-comment lines of source code (SLOC) in the system under test (SUT). The third column lists the number of non-blank, non-comment lines of source code in the system's test suite. The fourth column sums these quantities and the fifth column lists the amount of test code in the project as a percentage of the total. 57

3.2 Final production and test code size and increase in size over the two year study period. 60

3.3 Aggregated build results for 61 open source Java projects that use Travis and the state of 106,738 build executions from those projects. 61

3.4 Categorization of how builds transition between Pass and Fail states. The left-hand-side of → denotes the resources that changed to cause the test failure. The right-hand-side of → denotes the resources that were changed to resolve the test failure. This excludes flaky builds (177 transitions). . . . 67

3.5 The proportion of non-flaky Fail builds resolved by fixing faults and by maintaining the test suite. 68

3.6 Manually inspected tests as well as an example of a broken build and the change that fixed the test. The Break and Fix entries are hyperlinks to the original Travis-CI test suite execution output. 71

4.1 Investigated subject programs. 80

4.2 Number of candidate revisions, compilable revisions, and reproducible and isolated faults for each subject program. 82

4.3 Characteristics of generated test suites. 87

4.4 Number of real faults not coupled to mutants generated by commonly used mutation operators. Numbers are categorized by reason: weak implementation of a mutation operator, missing mutation operator, or no appropriate mutation operator exists. 94

4.5 Comparison of mutation scores between $\tilde{\boldsymbol{T}}_{\boldsymbol{pass}}$ and $\tilde{\boldsymbol{T}}_{\boldsymbol{fail}}$. 99

4.6 Comparison of studies that explored the relationship between mutants and real faults. 102

List of Figures

2.1 Usage of the 21,971 methods in the Android API. Each box represents a method. The area of the box represents the total number of times the method was called by all apps. The colour of the box indicates the API level in which the method was introduced. 18

2.2 Usage of the 21,971 methods in the Android API. Each box represents a method. The area of the box represents the number of apps in the corpus that call the method. The colour of the box indicates the API level in which the method was introduced. 19

2.3 The number of distinct Android API methods used by the apps in each of seven app corpora of different sizes. 22

2.4 The number of distinct Android API levels used by the apps in each of seven app corpora of different sizes. 23

2.5 The percentage of VirusTotal scanning tools that marked the apps we obtained from the Play store as malicious. 28

2.6 The percentage of VirusTotal scanning tools that marked the apps we obtained from malware corpora as malicious. 29

2.7 The distribution of Google Play apps among Android API levels. Colour indicates whether or not the app was assigned to the malicious category. . 31

2.8 The distribution of malicious apps from the Drebin, McAfee, and AMGP corpora among Android API levels. 32

2.9 The F-scores of k-nearest neighbour classifiers that used methods as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis. . . 38

2.10 The F-scores of k-nearest neighbour classifiers that used metrics as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis. . . 39

2.11 The F-scores of random forest classifiers that used methods as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis. 40

2.12 The F-scores of random forest classifiers that used metrics as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis. 41

2.13 The F-scores of the paired classifiers (randomly selected apps) as a function of the number of apps used to train and test the classifier. 43

3.1 Growth of production source code and test source code over the study period. At the beginning, test code accounted for only 10.8% of all code, but by the end this had increased to 21.9%, for a total of over 2.9 million lines of code. 60

3.2 Transitions between system states. Each transition is caused by one or more commits. Pass→Pass persisted for an average of 5.6 builds, Error→Error persisted for 2.6 builds, while Fail→Fail builds persisted for 2.9 builds. . . 62

3.3 As developers work on their systems their commits often change the state of the build. Each box represents a commit; builds are often not run on every commit but instead on blocks of commits. A C label on a commit means the code under test was changed; a T label means the test code was changed. The figure shows a failure caused by a code change that was resolved by fixing both code and test files. 65
3.4 Average difference between the number of code fixes and the number of test fixes. The radius of the bubbles represents the relative size of code+test fixes compared to all fixes. The sum of all code fixes and test fixes for a project is plotted on the x-axis on a logarithmic scale. 70
3.5 Proportion of test cases that fail for the 40 projects we were able to parse individual-test results from across the failing build of 586 tuples. Three data points (16.9%, 8.7% and 4.1%) have been elided for clarity. The average project failure rate was 0.38%. 73

4.1 Obtaining source code versions $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$. 83
4.2 Relationship between the $\boldsymbol{i}$-th obtained test suite pair $\langle \boldsymbol{T^i_{pass}}, \boldsymbol{T^i_{fail}} \rangle$ and the developer-written test suites $\boldsymbol{T_{bug}}$ and $\boldsymbol{T_{fix}}$. 84
4.3 Statement coverage ratios and mutation scores of the test suites $\boldsymbol{T_{pass}}$ for each subject program. 85
4.4 Statement coverage ratios and mutation scores of the generated test suites for each subject program. 88
4.5 Effect of triggering tests on mutant detection. 92
4.6 Snippets of real faults that require stronger or new mutation operators. . . 95
4.7 Snippets of real faults not coupled to mutants. 98
4.8 Correlation coefficients for each subject program. 100
4.9 $\boldsymbol{\hat{A}_{12}}$ effect sizes for mutation score differences between $\boldsymbol{\tilde{T}_{pass}}$ and $\boldsymbol{\tilde{T}_{fail}}$ for each subject program. 101

Chapter 1

Introduction

> *Nobody actually creates perfect code the first time around, except me. But there's only one of me.*
>
> *Linus Torvalds, 2007 [31]*

Maintaining and evolving modern software systems is a difficult task: their scope and complexity mean that seemingly inconsequential changes can have far-reaching consequences. Consider Debian's OpenSSL bug from 2008, when developers removed a line of code that caused the Valgrind and Purify tools to produce warnings about uninitialized memory usage [103]. One developer commented:

> *At lines 467-469 in crypto/rand/md_rand.c is an interesting thing:*

```
#ifndef PURIFY
MD_Update(&m,buf,j); /* purify complains */
#endif
```

> *That is the code that causes the problem (I just verified it with Valgrind). Does it have any bad side affects [sic] to always skip that code? Since both Purify and Valgrind is [sic] unhappy with that function call, something must be wrong with it. [14]*

Removing this line broke the random number generator, compromising all keys generated with this code.

There are many other cases where the poor practices used to effect software evolution caused or revealed serious faults. The Therac-25 disaster, where patients were given massive overdoses of radiation, is a widely studied example from the health domain [63]. In this case, a hardware failsafe was replaced by a software equivalent that contained a race condition. In 2009, a Google engineer mistakenly designated "/" as a site that was infected with malware, causing all websites to be flagged as malicious, including `google.com` itself [75]. Poorly executed software evolution practices have also been identified as the cause of four serious performance bugs in the Linux scheduler [69].

In all of these examples, the scope and complexity of the systems prevented the developers from fully understanding the ramifications of their changes. Most software development companies institute processes to help developers manage this complexity when making changes in order to prevent, as much as possible, the introduction of faults. In their most basic form, these processes are based on intuition, industry trends, or the experience of a limited number of developers. An alternative approach is to measure descriptive software metrics, such as the number of lines of code in the system, the number of faults found in each source code file, the percentage of code that is executed by the test suite, and so on. Development processes can be designed to keep these metrics in a predefined range; for example, management may insist that test suite coverage is at least 75% prior to product release. However, this approach can do as much harm as good: metrics that seem intuitive may not have empirical support [43] or, when metrics are tied to compensation, developers may game the metrics, to cite two potential problems.

A more sophisticated approach is to use *data science* to create robust, empirically-based development processes. Data science is a set of techniques for extracting actionable insights from (usually large) datasets. It arose as a way to handle the "big data" problem that has recently manifested itself in many areas, such as business analytics and health care informatics. Data science incorporates methods from fields that have historically been separate, such as machine learning, statistics, databases, and distributed systems. The latter two areas in particular come into play as the volume of data increases to the point that it can no longer be stored on a single computer.

Software development generates many artifacts, including revision control history, issue reports, electronic communication records, and software documentation. Software engineering researchers can apply data science techniques to these data sources to extract testable research insights that can guide the creation of development processes. The application of data science techniques to software engineering data has been considered in several prior PhD theses, notably those of Zimmerman [122] and Hassan [38]. In software engineering research, the use of data science techniques is sometimes referred to as "mining software repositories" or "software analytics".

Of course, data science is not the only approach to software engineering research: other methods of inquiry such as case studies and developer surveys can provide valuable information. However, data science provides a way of summarizing, linking, and merging large volumes of data from many disparate sources to address real questions that developers have about their systems. This is not information that can be extracted or inferred using other techniques such product-based static program analysis or simple textual tools such as `grep`. For example, Herzig et al. [40] used data science techniques at Microsoft to determine how to run fewer regression tests while minimizing the number of faults that were not detected as a result. Ostrand et al. [85] predicted the location and number of faults in two large industrial systems. We have previously shown that different software artifacts such as the issue tracker, the documentation, and Stack Overflow posts can be linked via the fully-qualified names of source code elements [45].

This dissertation presents three examples of how data science techniques can be used to generate insights about the development of software systems, in aid of improved development practices and overall system quality. To be precise, the thesis of this dissertation is as follows:

Thesis Statement. *Robust development processes are necessary to minimize the number of faults introduced when evolving complex software systems. These processes should be based on empirical research findings. Data science techniques allow software engineering researchers to develop research insights that may be difficult or impossible to obtain with other research methodologies. These research insights support the creation of development processes. Thus, data science techniques support the creation of empirically-based development processes.*

We provide evidence for this thesis by presenting three research insights that could not have been obtained without the use of data science. Note, however, that this dissertation does not investigate the creation of development processes based on these insights.

First, we present insights into automated malicious Android application (app) detection. Many of the prior studies done on this topic used small corpora that may provide insufficient variety to create a robust app classifier. Currently, no empirically established guidelines for corpus size exist; in practice, previous studies have used anywhere from tens of apps to hundreds of thousands of apps to draw their conclusions. This variability makes it difficult to judge if the findings of any one study generalize. We attempted to establish such guidelines and found that 1,000 apps may be sufficient for studies that are concerned with common app behaviour, for instance the most popular Android API methods, while more than a million apps may be required in studies that want to identify outliers. Moreover, many prior studies of malicious app detection used old malware corpora in their experiments

that, combined with the rapid evolution of the Android API, may have influenced the accuracy of the studies. We investigated this problem by studying 1.3 million apps and showed that the evolution of the API does affect classifier accuracy, but not in the way we originally predicted. We also used our API usage data to identify the most infrequently used API methods. The use of data science techniques allowed us to study an order of magnitude more apps than previous work in the area; additionally, our insights into infrequently used methods illustrate how data science can be used to guide API deprecation. Chapter 2 of the thesis describes this work in detail.

Second, we present insights into the costs and benefits of regression testing. Regression test suites grow over time, and while a comprehensive suite can detect faults that are introduced into the system, such a suite can be expensive to write, maintain, and execute. These costs may or may not be justified, depending on the number and severity of faults the suite can detect. By studying 61 projects that use Travis CI, a continuous integration system, we were able to characterize the cost/benefit tradeoff of their test suites. For example, we found that only 74% of non-flaky test failures are caused by defects in the system under test; the other 26% were caused by incorrect or obsolete tests and thus represent a maintenance cost rather than a benefit of the suite. Data about the costs and benefits of testing can help system maintainers understand whether their test suite is a good investment, shaping their subsequent maintenance decisions. The use of data science techniques allowed us to study a large number of projects, increasing the external generalizability of the study and making the insights gained more useful. Chapter 3 of the thesis describes this work in detail.

Third, we present insights into the use of *mutants* to replace real faulty programs in testing research. Mutants are programs that contain deliberately injected faults, where the faults are generated by applying *mutation operators*. Applying an operator means making a small change to the program source code, such as replacing a constant with another constant. The use of mutants is appealing because large numbers of mutants can be automatically generated and used when known faults are unavailable or insufficient in number. However, prior to this work, there was little experimental evidence to support the use of mutants as a replacement for real faults. We studied this problem and found that, in general, mutants are an adequate substitute for faults when conducting testing research. That is, a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed, for both developer-written and automatically-generated test suites. However, we also found that additional mutation operators should be developed and some classes of faults cannot be generated via mutation. The use of data science techniques was an essential part of generating the set of real faults used in the study. Chapter 4 of the thesis describes this work in detail.

Taken together, the results of these three studies provide evidence that data science is a useful set of techniques that allow software engineering researchers to develop insights that would previously have been unobtainable.

1.1 Roles and Publication Status

Chapter 2 is independent work. It has been reformatted as a paper and is currently in submission at ICSME 2017 [44].

Chapter 3 is joint work with Adriaan Labuschagne. It has been reformatted as a paper and is currently in submission at FSE 2017 [61]. My role in the work included data analysis and writing.

Chapter 4 is joint work with René Just and Darioush Jalali [53]. It was published at FSE 2014 and won an ACM Distinguished Paper Award. My role in the work included conceptualizing the study, the generation of test suites, data analysis, and writing.

The material in Chapter 4 is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Some of the computations in this work were performed using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: `www.sharcnet.ca`).

Chapter 2

Detecting Android Malware

Smartphones are becoming increasingly popular, with the two primary smartphone operating systems being Apple's iOS and Google's Android. Of these, Android is more widespread, commanding over 80% of the global mobile OS market share [28]. This popularity makes it a particularly attractive target for malware developers, who have begun to expand from desktop software to mobile applications (apps). In the third quarter of 2014, McAfee reported that they had detected over 5 million mobile malware samples. That number grew by 16% during that quarter and by 112% during the previous year [72]. Trend Micro reported a figure of 7.1 million malware samples in the first half of 2015 [76]. Malicious apps will only become more common as smartphones continue to replace traditional computers and cell phones and the use of third party apps for sensitive purposes such as mobile banking becomes routine.

The increase in malware implies a need for accurate mobile malware detection tools. Seventeen previous studies have used machine learning to identify malicious Android apps with promising results, as we will discuss in Section 2.1. However, these studies share two issues. First, no empirical guidelines exist regarding how many apps should be used in studies of Android to obtain a generalizable result. This leads to the use of corpora of widely varying sizes; the previous studies in this area used between 436 and 135,792 apps. While more is generally accepted to be better, app mining can be expensive; in addition, the lack of guidelines makes it harder to judge the generalizability of the findings in each paper.

The second issue that was common to previous work in the area is that the studies did not control for the API level[1] of the app, even though the studies used level-dependent

[1] Android API developers use the term "version" in the usual way. The term "level" is used to refer to

input features, such as the Android API methods called by the app. This is an issue because it is difficult to collect malicious apps: identifying malware in the wild is time consuming and requires special knowledge, so once a good quality malware corpus has been curated, it is often used by researchers for some time. Consequently, the benign apps in a study are often newer – and thus use a higher API level – than the malicious apps. Combined with the fact that the classifiers used level-dependent input features, the classifiers in previous work may have been distinguishing old apps from new apps rather than malicious apps from benign apps.

To address these two threats, we collected 1,368,376 Android apps and conducted two studies. First, we explored how the results obtained in a mining-based study vary depending on the number of apps used. Specifically, we asked the following research question:

Research Question 2.1. *How many apps should a mining-based study use to obtain a generalizable result?*

To answer this question, we studied various features of seven app corpora of varying size, ranging from 10 apps to 1,368,376 apps, as we will describe in Section 2.2.1. We found that, as a "rule of thumb", 1,000 apps may suffice when the study is concerned with what the majority of apps do. Studies that attempt to detect outliers of any sort, such as malware, naturally require more apps, and it seems that 1,000,000 or more may be required. Section 2.2.2 presents these results in detail.

Second, we explored the influence of API level on the accuracy of malware detection classifiers. Specifically, we asked the following research question:

Research Question 2.2. *Should studies of Android malware detection control the API level of the apps used in the study?*

We assigned the Android apps we collected to the API levels 1 through 22, inclusive, based on their usage of Android methods and permissions. We trained and tested classifiers on the apps assigned to each level, then trained and tested classifiers on "paired" collections of apps from all levels. In each case, the paired classifier was trained and tested using the same number of apps as the corresponding level-controlled classifier. We compared the performance of the level-controlled classifiers to the paired (non-level-controlled) classifiers to determine if controlling API level decreased classifier accuracy. The level-controlled

a group of one or more consecutive versions that are very similar to each other. Thus, "level" is a more coarse-grained term.

classifiers thus represent the experimental case while the paired classifiers represent the control case, and the fact that each pair is trained and tested with the same number of apps ensures that corpus size does not influence the comparison. Section 2.3.1 describes our procedure in detail. We found that controlling app API level, contrary to our hypothesis, *increased* classifier accuracy. This implies that controlling API level when evaluating malware detection classifiers is not necessary. Section 2.3.2 presents these results in detail.

Following the presentation of results, Section 2.4 discusses how our findings can be used by both researchers and Android API maintainers. Section 2.5 presents threats to validity and Section 2.6 explains how the work in this chapter supports our thesis statement.

2.1 Related Work

We discuss work related to our two research questions in turn.

2.1.1 Corpus Size Guidelines

We were unable to identify any studies that answered our specific research question. Though Martin et al. [71] studied the effect of using a small sample of app *reviews*, they did not consider the sampling problem when the apps themselves are studied. Nagappan et al. [78] developed a metric for measuring the diversity of software projects, but the existence of such a metric does not provide an easy-to-use guideline. In addition, their metric is for general software engineering studies, not just those done with Android.

Traditional statistics can answer our question in some cases. For example, when estimating the population mean from a sample mean, one can use the following equation:

$$n = \frac{z^2\sigma^2}{E^2}$$

where z is the z-score required to get a particular confidence value, σ is the standard deviation of the population, and E is the error of estimation [26]. Concretely, to estimate the mean of a population with a standard deviation of 0.2 units ($\sigma = 0.2$) within 0.05 units of the population mean ($E = 0.05$) with a confidence of 99% ($z = 2.576$), one should use a sample of at least 107 observations. Unfortunately, computations of this nature have limitations. The most salient is that they are limited to the calculation of numerical values such as the mean; statistics does not tell you how many apps you must examine if you want to know which API methods are most popular. In addition, such computations

rely on assumptions made about the population distribution. If a software engineering researcher wanted to study, for instance, the average number of lines in a stack trace, it is not necessarily the case that this value is normally distributed and, if it is, what the standard deviation of the distribution is.

Thus, to the best of our knowledge, guidelines of the sort in this work have not been previously proposed. However, four papers have studied the usage of Android methods; these papers are relevant to the discussion in Section 2.4 of how API maintainers can use our results.

Li et al. [65] identified the ten most popular API methods used by a sample of 100,000 malicious Android apps. The authors then used static analysis to identify the most common string parameters passed to these methods. They found that this information could be used to identify some of the apps as malicious, as some apps that send SMSs to premium numbers have the phone numbers hardcoded into the app. The authors also suggest that information about common parameters could be used to support API comprehension and could be used to find variants of a given app (e.g., plagiarized apps). However, the authors did not evaluate the tool in these two scenarios.

Li et al. [66] studied how Android apps use so-called "inaccessible" API methods, or methods not intended for use by third party apps. Using a corpus of 23,666 apps, mainly collected from Google Play, the authors found that 5% of the apps used these inaccessible API methods. The apps used a median of two inaccessible methods each and these methods were mainly related to `view` and Bluetooth functionalities. Over 56% of the inaccessible methods had a lifecycle of only one version; that is, they were removed from the Android framework in the version after the one in which they were introduced.

McDonnell et al. [73] sought to understand how Android app developers respond to changes made to the Android API. They studied the version control repositories of ten open-source apps and found that 28% of API references in these apps were outdated; that is, the app developers called methods that had been replaced by newer versions of the same methods in the API. In addition, 50% of these outdated references lagged by at least 16 months, meaning that there was a 16 month gap between the time a new method was introduced in the API and the time the developer updated their own code to use the new method. However, the authors did not take multi-version support into account: if the app developer wants to support phones running older versions of Android, this lag and the use of older API levels in general may be deliberate.

Kavaler et al. [57] studied the relationship between the popularity of the classes in the Android API and the number of Stack Overflow questions and answers about the classes. They measured the 'popularity' of a class by collecting a large corpus of over 100,000

apps and counting the number of times the methods of each Android class were called by the apps. The authors found that the popularity of a class was correlated with the number of Stack Overflow questions asked about that class; that the size of a class in lines of code was correlated with the number of questions asked about that class; and that classes with more documentation also had more questions asked about them. Note that these findings are correlations, not casual explanations; for example, it could be that when classes are frequently asked about on Stack Overflow, the Android developers improve the documentation for those classes; alternatively, it could be that a developer faced with an overwhelming amount of documentation turns to Stack Overflow for a fast, personalized answer.

2.1.2 Malware Detection

We searched the literature for papers that do the following:

- Attempt to classify Android apps as malicious or benign (allowing for a possible third category, adware, or multiple additional categories if the classifier attempts to identify the malware family);
- Perform classification using a machine learning classifier;
- Create feature vectors using static analysis only; and
- Execute the classifier on either the mobile device or a central server.

We focused on papers that use static analysis because we are interested in papers that might use features specific to a particular Android API level, such as the methods and permissions used by the app.

We identified 21 studies that met the inclusion criteria. Of these, 17 studies simply attempt to detect malicious apps using a classification approach. The authors' aim was to produce a classifier that has high precision and recall. The other four are 'meta-studies' of sorts, similar to this work, that attempt to determine how different research approaches affect the accuracy of the resulting classifiers. These papers are thus not concerned with maximizing classifier accuracy but rather with the relative accuracy of classifiers produced using different methods. We discuss these two types of papers in turn.

Studies of App Classification

Our second research question asks whether classifier accuracy drops when the API level of the apps is controlled. However, answering this question is valuable only if existing classifiers use API-level-dependent input features: if not, there is no reason to answer the research

question. Thus, we explored the classification features used by the seventeen previous studies of Android malware detection. Of these seventeen, thirteen studies explicitly use Android method calls, Android permissions, or both as input features when training the classifier [1, 7, 8, 27, 29, 30, 99–101, 112, 113, 115, 116]. Two use dataflow paths within the app, implicitly relying on method calls [9, 64]. One uses only the string constants within the app [102], and the final study uses code complexity metrics computed for the methods and classes of the app [92].

The fact that fifteen of these seventeen studies use feature vectors that depend either explicitly or implicitly on the API level of the app indicates that it is important to determine whether controlling for API level affects classifier accuracy. However, we note that the final two studies, whose input features did not depend on the API level, obtained fairly good results. Sanz et al. [102] achieved a true positive rate of 94% at a false positive rate of 9% with a random forest classifier. Protsenko and Müller [92] achieved a true positive rate of 94% at a false positive rate of 0.5% with a random forest classifier. This implies that a classifier does not necessarily need to use API-level-dependent input features to correctly classify apps.

Meta-Studies of App Classification

Four studies have considered whether the research methods used by security researchers—or, more accurately, the assumptions underlying the methods—affect the accuracy of app classification. Most similar to our work is a paper by Roy et al. [98] that considered the impact of common methodological choices on classifier accuracy. In particular, the authors considered whether using an outdated malware corpus can affect the performance of the classifier. The authors trained two classifiers using the same benign apps but different malicious apps. The first classifier used the Android Malware Genome Project (AMGP) corpus as the malicious apps; this corpus was created in 2012 [121]. The second classifier used new malware from Google Play as the malicious apps. The authors found that the performance of the classifier that was trained and tested on new malicious apps was worse than the other classifier. However, the authors did not investigate the cause of the difference, and there are a number of plausible explanations. First, the new corpus could contain more diverse malware samples. The AMGP corpus contains 49 different malware families, and while this may have been the majority of families that existed at the time the corpus was created, to date over 300 families have been identified [106]. A second possibility is that the new malware is more sophisticated than the older malware and thus harder to detect. A third possibility is that the API level of the apps was a confounding variable; i.e., the first classifier was distinguishing old apps from new apps rather than malicious apps from

benign apps. Our paper differs from Roy et al. in that we seek to determine if this third explanation is the root cause of the difference they observed.

Allix et al. [2, 3] noted that, in practice, a classifier is trained on whatever malware is currently available but will be used (i.e., tested) on malware that will be written in the future. They therefore argued that the most common method of building and evaluating classifiers, where apps are randomly assigned to the testing and training sets, produces biased results: it allows the classifier to be trained on apps "from the future". That is, random selection means that some of the apps in the test set can be historically antecedent to some of the apps in the training set. Giving the classifier knowledge of the future in this way may make its accuracy unrealistically high. The authors tested this hypothesis by training a random forest classifier with both *historically coherent* datasets, where the apps in the training set were older than those in the test set, and random datasets, where the apps in the training and test sets were selected randomly. They found that the classifier that used the historically coherent training set always had a lower F-score than the classifier that used the random training set. Moreover, for the historically coherent classifier, the wider the time gap between the apps in the training set and the apps in the test set, the worse the classifier performed. For example, when the classifier was trained on apps from August 2011 and tested on apps from August 2013, the F-score was less than 0.1.

These results may seem to contradict those of Roy et al., since Allix et al. found that a wider age gap between the training and test sets *decreased* classifier accuracy. However, the two studies were considering slightly different problems. In Roy et al.'s study, the benign apps were new in both the control and experimental conditions; only the malicious apps were varied. Thus, the authors found that when malware is made "more different" from the benign apps by introducing an age gap, classifier accuracy improved. Allix et al. were manipulating app age for both the malicious and benign apps. Thus, the authors found that when the classifier was trained on old apps, the features that were good predictors for those apps were no longer good predictors for the new apps.

In this study, we hypothesized that both results could be explained by Android API churn. The Android API changes quickly: every month, on average, 44 methods are changed, 11 methods are added, 51 fields are changed, 9 fields are added, and less than one method/field is removed [73]. In Roy et al.'s study, the old malware would have used different Android methods than the new malware and new benign apps, making the old apps easier to detect. In Allix et al.'s study, the greater the age gap between the testing and training sets, the more the API would have changed; consequently, the input features that were good predictors for the training set would no longer be good predictors for the test set.

Allix et al. [4] later performed another study to determine if classifiers that perform well “in the lab” will also perform well “in the wild”. While they investigated a number of different classifier properties, such as the number of features used and the malware to benign app ratio used, the primary difference between the “lab” scenario and the “wild” scenario was the number of apps used to train and test the classifiers. The “in the lab” scenario used AMGP as the malicious apps and apps from Google Play as the benign apps; this classifier used at most 4,747 apps for both training and testing. The “in the wild” scenario trained the classifier using the entire universe from the “in the lab” study, i.e., the 4,747 AMGP and Play apps. The classifier was then tested on a large number of apps from Google Play. The exact number ranged from 48,422 apps to 51,302 apps; the malicious or benign nature of the apps was determined using VirusTotal. The authors did not indicate the detection threshold they required from VirusTotal, but mentioned that 16% of the Play apps were labelled as malware. The “wild” classifiers performed uniformly worse than the “lab” classifiers, no matter how the other variables, such as number of features, were varied. This indicates that having a wide variety of apps in a corpus makes classification more difficult. However, it is not clear from their results if the sheer number of apps is the important variable, or the fact that the “lab” classifier was trying to identify malware from AMGP (old malware) while the “wild” classifier was trying to identify malware from Google Play (new malware), which is essentially the scenario studied by Roy et al.

2.2 RQ1: How Many Applications?

In this section, we explore our first research question: how many apps should a mining-based study use to obtain a generalizable result? We discuss the method we used to answer this question in Section 2.2.1 and our results in Section 2.2.2.

2.2.1 Method

Our general approach was to determine how large a corpus needs to be before its characteristics stabilize. That is, we measured various features of app corpora of varying sizes, then explored how those measurements changed with corpus size and how large the corpus needed to be before the measurements stopped changing. To accomplish this, we needed to construct app corpora of varying sizes, and we needed to measure various features of those corpora. We describe these two steps in turn.

Creating App Corpora

We began by downloading 1.5 million Android apps from the PlayDrone snapshot of Google Play [109]. The apps were selected randomly from the crawled set of apps in order to avoid biasing the selection. As of May 3, 2016, there were 2,126,544 apps in the Google Play store[2], so our corpus represents a large part of the store. We ensured we did not include duplicate apps by computing the SHA-256 hash of each app; apps with distinct hashes were assumed to be different. We created seven corpora of the following sizes: 10 apps, 100 apps, 1,000 apps, 10,000 apps, 100,000 apps, 1,000,000 apps, and 1,368,376 apps. The largest corpus did not include all 1.5 million apps due to our callgraph extraction tool failing on some apps; see Section 2.2.1. Each corpus was a subset of the next larger corpus.

Measuring App Features

We chose to compare the corpora using the following four features:

- the most popular 20 Android API methods used by the apps in the corpus;
- the least popular 20 Android API methods used by the apps in the corpus;
- the number of distinct API methods used by each app in the corpus; and
- the number of distinct API levels (versions) used by each app in the corpus.

We selected these features because they are commonly considered in studies of Android. For example, the four papers discussed in Section 2.1 [57, 65, 66, 73] all consider the usage of Android methods. The studies that attempt to detect Android malware based on the usage of API methods also naturally rely on the identification of popular methods. Other examples include studies by Parnin et al. [88] and Linares-Vásquez et al. [67, 68].

To compare the features we selected, we needed to construct a master list of Android API methods and the API level in which each was introduced and determine which methods were used by each app. We discuss these steps in turn.

Master Method List We obtained a list of Android methods and the API level in which they were introduced by parsing files from the official Android developer documentation. To get the methods that existed in level 1, we parsed the `api-versions.xml` file. We looked for classes with the attribute "since" equal to 1, and within those classes looked for any method that also had an attribute "since" equal to 1. We mapped all of these methods to level 1 of the API.

[2] `http://www.appbrain.com/stats/stats-index`

We then parsed the `api-diff` files that are provided by the Android developers for levels 3 through 22; level 2 does not have any diff files available. The diff files for level n list all changes to the API made between level $n-1$ and n. For every level, we parsed the change files to find methods that were added in level n and assigned them to API level n in our list of methods. In total, we identified 21,978 different methods in the 22 levels. Table 2.1 summarizes the number of methods in each API level. Note that level 2 of the API does not have any `api-diff` files available; we therefore have no methods mapped to that level. Level 6 of the API did not add any methods, so we also have no methods mapped to level 6. Note also that we considered all public methods included by the Android team in their diff reports to be part of the API. This means that, for example, `java.lang.String.length()` and `org.apache.http.StatusLine.getStatusCode()`, methods one might not instinctively think of as Android API methods, are included. We include these methods because the Android developers have copied or re-implemented this functionality in their own code base, and consequently must maintain this code over time.

Determining Method and Level Usage Next, to determine which API methods were used by a given app, we used the Androguard[3] tool to generate a callgraph for each app. We were able to extract callgraphs for 1,368,376 of the apps we collected from the PlayDrone snapshot. We parsed these callgraphs to identify the methods used by the app and removed non-Android methods using our list of Android methods. For each app, we counted the number of distinct Android methods it called and the number of distinct API levels it used, where a level is "used" if the app calls a method that was introduced in that level. Concretely, if an app uses methods that were introduced in API levels 1, 10 and 12, we consider that it uses three distinct API levels. We then combined the results from all apps to determine the most and least popular methods for each corpus. When determining popularity, we counted *all* method calls and not just distinct calls for each app; that is, if a given app calls a method ten times, we increment the total number of calls for that method by ten. Figures 2.1 and 2.2 clarify this difference. The figures are treemaps, where each box represents one method; the colour of each box represents the level in which it was introduced. In Figure 2.1, the area of each box is proportional to the total number of times each method is called. In Figure 2.2, the area of each box is proportional to the number of distinct apps calling the method. Note that API methods that were not used by any of the apps in the corpus have zero area and are therefore not represented in the figures.

[3] `https://github.com/androguard/androguard`

Table 2.1: The number of Android methods introduced and removed in each API level (version) and the total number of methods in each level.

Level	Methods added	Methods removed	Total
1	17,627	0	17,627
2	0	0	17,627
3	417	0	18,044
4	86	3	18,127
5	205	0	18,332
6	0	0	18,332
7	53	0	18,385
8	208	2	18,591
9	529	17	19,103
10	7	1	19,109
11	449	10	19,542
12	79	2	19,619
13	22	0	19,641
14	302	17	19,926
15	27	0	19,953
16	393	10	20,336
17	151	19	20,468
18	159	4	20,623
19	287	4	20,906
20	32	1	20,937
21	859	17	21,779
22	79	2	21,856

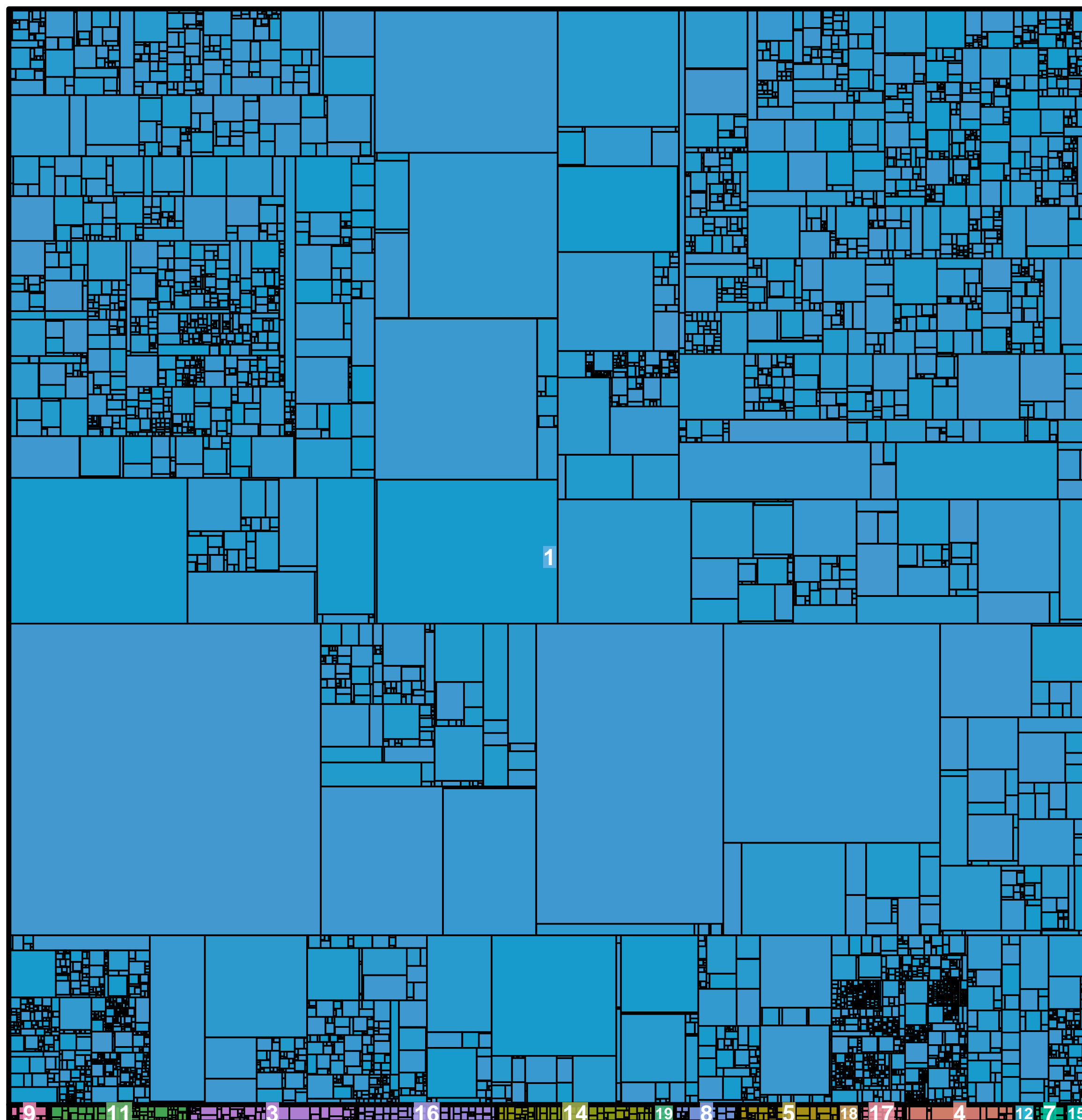


Figure 2.1: Usage of the 21,971 methods in the Android API. Each box represents a method. The area of the box represents the total number of times the method was called by all apps. The colour of the box indicates the API level in which the method was introduced.

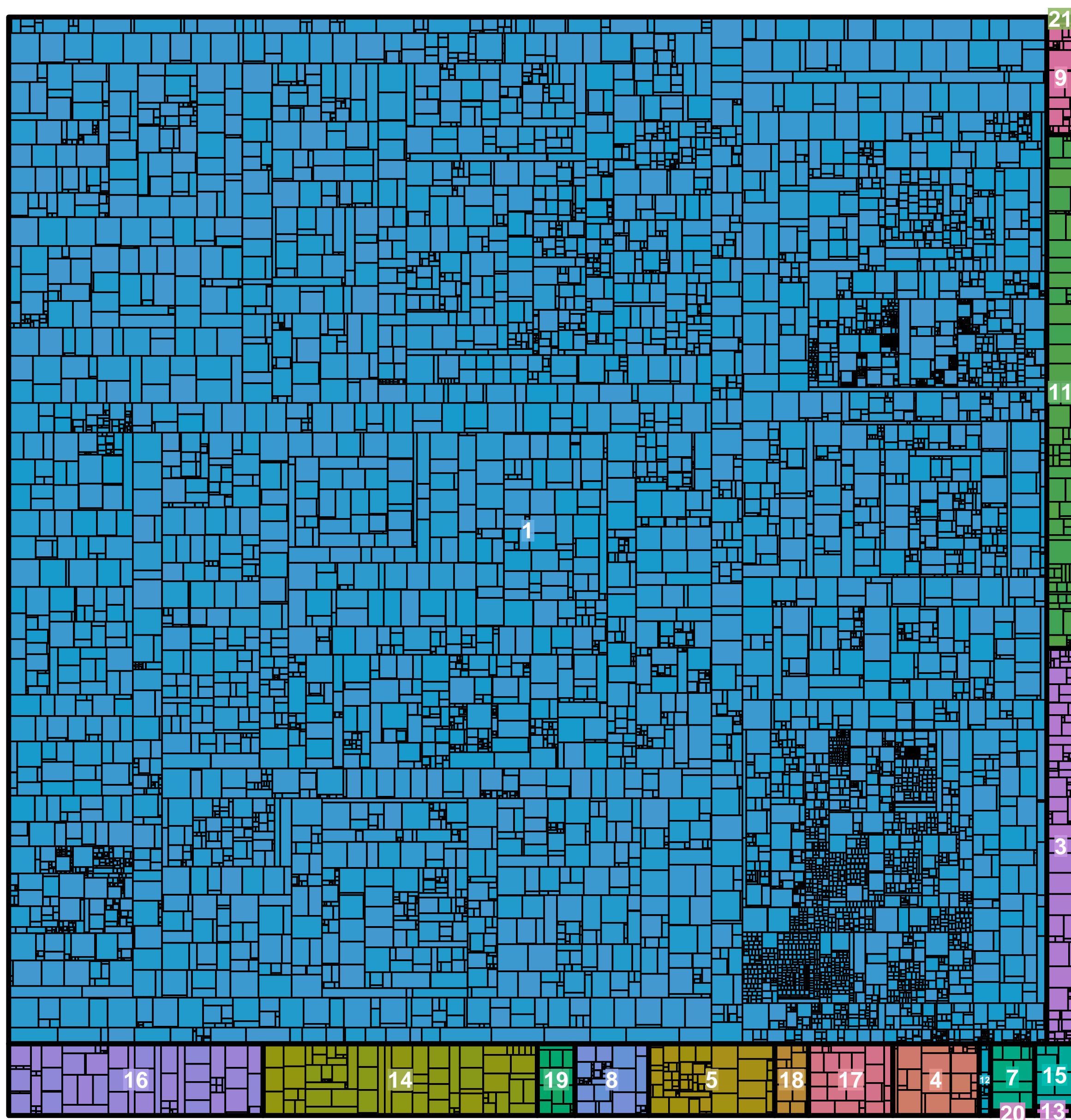


Figure 2.2: Usage of the 21,971 methods in the Android API. Each box represents a method. The area of the box represents the number of apps in the corpus that call the method. The colour of the box indicates the API level in which the method was introduced.

Generating our dataset was an extremely time consuming process. To manage this for our 1,368,376 apps, we employed Computer Canada's Sharcnet research cluster which provides multiple 1,000+ core clusters for sequential jobs. These jobs generated 11 TB of callgraphs that each had to be extracted, parsed, and combined to achieve our final dataset. This supports our assertion in the introduction of this chapter that app mining is expensive and further motivates our desire to understand how many apps are necessary for generalizable results.

2.2.2 Results

We begin our presentation of the results by comparing the number of distinct methods and levels used by the apps in each corpus. Figures 2.3 and 2.4 show this information as boxplots. In Figure 2.3, the median and first and third quartiles are fairly stable for all corpora containing at least 100 apps. However, the largest corpora have more outliers.[4] In Figure 2.4, we see roughly the same pattern; however, the first and third quartiles do not stabilize until the corpus contains 1,000 apps and the median does not stabilize until the corpus contains 10,000 apps.

More formally, we can compare the distributions of pairs of corpora using statistical tests. We compare each of the five smallest corpora to the corpus containing one million apps. When doing each comparison, our null hypothesis is that the data sets are drawn from the same population distribution function. As we know that all apps are drawn from the same universe of apps, disproving the null hypothesis indicates that—to be extremely imprecise—the smaller corpus is "very different from" the one-million-app corpus.

We use two statistical tests: the two-sample Kolmogorov-Smirnov (K-S) test and the two-sample Kuiper test. The K-S test is most sensitive around the median value [91]. Roughly speaking, it is good at detecting distribution shifts but may not be able to distinguish between distributions that vary only in the outlying values. The Kuiper test is a variant of the K-S test that accounts for this and so is equally sensitive at all parts of the distribution. All this means that, if we reject the null hypothesis for *both* tests, neither the central values nor the outliers of the smaller corpus are representative of the one-million app corpus. If we reject the null hypothesis with the Kuiper test but not the K-S test, the central values are representative of the one-million-app corpus but the outlying values are not. If we cannot

[4] By default, R plots a data point as an outlier if it is more than $1.5 * \mathrm{IQR}$ above the third quartile or more than $1.5 * \mathrm{IQR}$ below the first quartile, where IQR is the interquartile range, or the difference between the third and first quartiles.

reject *either* null hypothesis, the two data sets are consistent with a single distribution function; that is, the smaller corpus is representative of the one-million-app corpus.

Table 2.2 shows the p-values obtained from this test for the distinct Android methods used and the distinct Android levels used. For the Android methods used, we see that we cannot reject the null hypothesis for the corpora of size 100 and size 100,000. That is, the entire distribution of the number of methods used for those two corpora, including outliers, is similar to the distribution for the one-million-app corpus. However, given that we can reject the null hypothesis with $p = 0.06$ and $p = 0.01$ for the 1,000 and 10,000 app corpora, our results suggest that one needs a corpus of 100,000 to adequately represent a population of 1,000,000 apps, including outliers. As the Play store has several million apps at the time of this writing, one might consider using several hundred thousand apps. We cannot reject the null hypothesis for any of the corpora when using the K-S test, indicating that the central values of the distributions of the two data sets are similar even when we are comparing a 10-app corpus to the one-million-app corpus. However, given that the p value of the K-S test for the 10 app corpus is only 0.28, one may prefer to use 100 apps in place of the larger corpus instead of 10 apps.

For the Android levels used, we can only reject the null hypothesis for the corpus of 10,000 apps using the Kuiper test. In other words, all other corpora were representative of both the outliers and the central values of the one-million-app corpus, even the corpus containing only ten apps.

Next, we compare the most and least popular 20 API methods in each corpus. We consider the most and least popular 20 methods in the largest corpus, with 1,368,376, as the gold standard, and compare the most and least popular methods in the other corpora to those results. Table 2.3 shows how many of the top 20 methods in the smaller corpora match the top 20 in the largest corpus, both when rank order is considered and when it is not. Table 2.4 shows the same information for the 20 least popular methods including methods with no calls; that is, all of the methods considered were not used by the apps in the corpus. Table 2.5 shows the same information for the 20 least popular methods that are called by at least one app. As the tables show, the top 20 methods are extremely similar in all corpora when order is ignored. When order is considered, using at least 1,000 apps results in an 80% match with the largest corpus. For the bottom 20 methods with zero calls, however, good results are not obtained until 1,000,000 methods are used. For the bottom 20 methods called by at least one app, even a corpus of 1,000,000 apps does not produce good results: only 35% of the methods matched the least popular methods in the largest corpus even when order was ignored, and only 10% matched when order was considered.

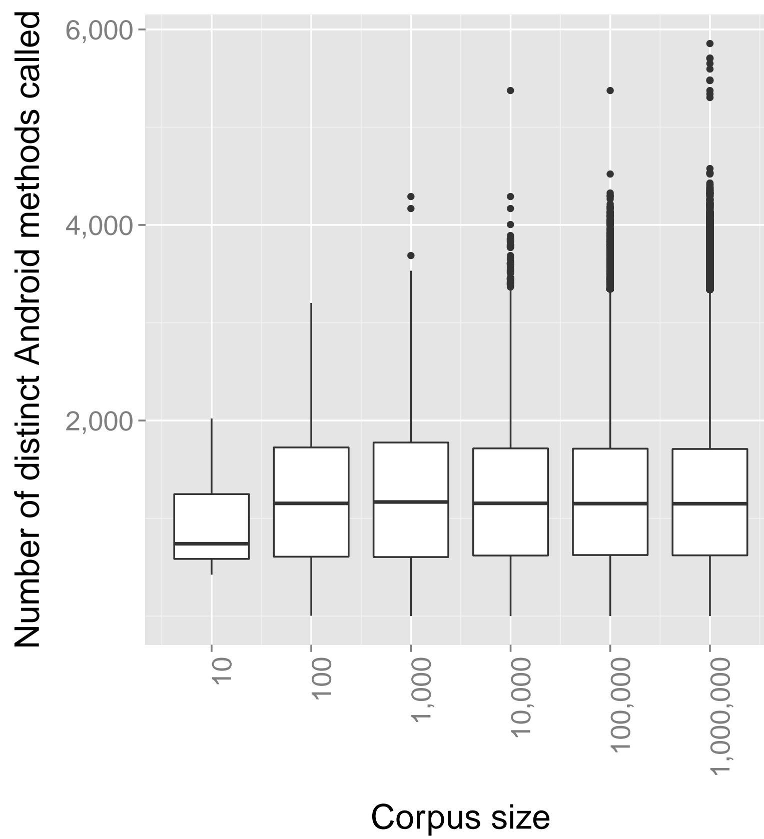


Figure 2.3: The number of distinct Android API methods used by the apps in each of seven app corpora of different sizes.

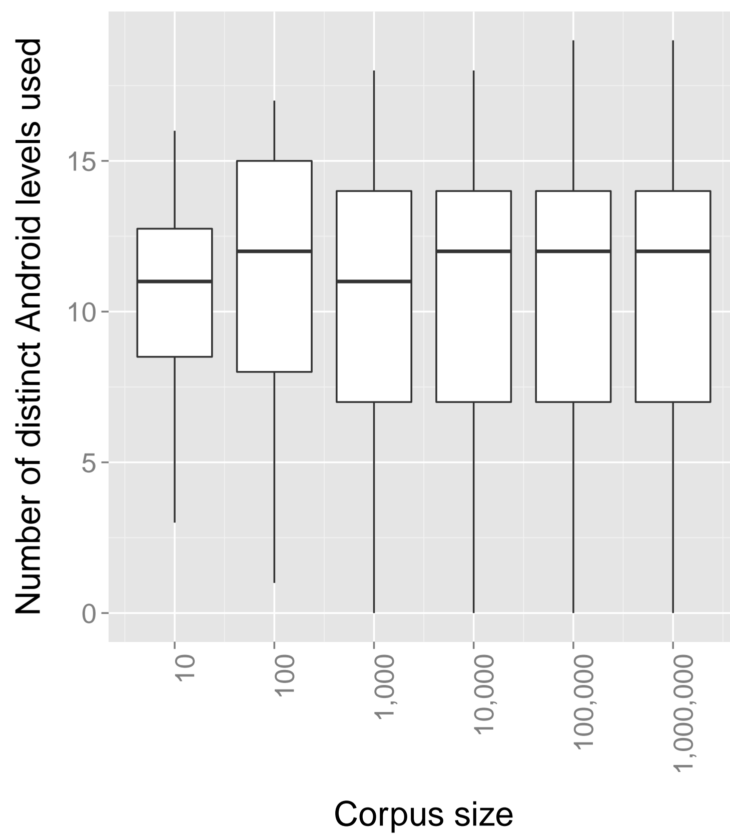


Figure 2.4: The number of distinct Android API levels used by the apps in each of seven app corpora of different sizes.

Table 2.2: P values of the Kolmogorov-Smirnov (K-S) and Kuiper tests comparing each smaller corpus to the largest corpus (1,000,000 apps). The null hypothesis is that the data sets are drawn from the same population distribution function. The alternative hypothesis is that the two experimental distributions are consistent with the same theoretical distribution. The second and third columns give P values for the distributions of distinct methods used, while the fourth and fifth columns give P values for the distributions of distinct levels used.

Corpus Size	K-S (DM)	Kuiper (DM)	K-S (DL)	Kuiper (DL)
10	0.28	0.01	0.99	0.92
100	0.83	0.45	0.97	0.86
1,000	0.46	0.06	0.91	0.65
10,000	0.28	0.01	0.28	0.01
100,000	0.97	0.84	0.64	0.19

Table 2.3: The percentage of the most popular 20 API methods in each corpus that match the most popular 20 API methods in the largest corpus, both when order is considered and when it is ignored.

Corpus size	% matches (rank considered)	% matches (rank ignored)
10	20	90
100	60	100
1,000	80	100
10,000	80	100
100,000	100	100
1,000,000	100	100

Taken together, our results suggest that between 10 and 1,000 apps are necessary to capture typical behaviour, depending on the corpus feature of interest. A million or more apps may be required to capture outlying values, again depending on the corpus feature of interest. We combine these results in the following guideline.

Answer 2.1. *A corpus containing 1,000 apps is likely sufficient for studies that consider "normal" behaviour, or things that the majority of apps do. Detecting outliers naturally requires more apps, and over a million may be required for a generalizable study.*

We discuss the results further in Section 2.4 after we answer our second research question.

Table 2.4: The percentage of the least popular 20 API methods in each corpus that match the least popular 20 API methods in the largest corpus, both when order is considered and when it is ignored, including methods with zero calls.

Corpus size	% matches (rank considered)	% matches (rank ignored)
10	0	15
100	5	30
1,000	10	40
10,000	10	50
100,000	20	60
1,000,000	100	100

Table 2.5: The percentage of the least popular 20 API methods in each corpus that match the least popular 20 API methods in the largest corpus, both when order is considered and when it is ignored, excluding methods with zero calls.

Corpus size	% matches (rank considered)	% matches (rank ignored)
10	0	0
100	0	0
1,000	0	0
10,000	0	0
100,000	0	0
1,000,000	10	35

2.3 RQ2: Should API Level Be Controlled?

In this section, we explore our second research question: should studies of Android malware detection control the API level of the apps used in the study? We discuss the method we used to answer this question in Section 2.3.1 and our results in Section 2.3.2.

2.3.1 Method

To answer our research question, we used the following high level procedure:

- Gathered apps from Google Play and used the VirusTotal service to determine their benign or malicious nature;
- Gathered additional malicious apps from established corpora to enhance our own corpus;
- Assigned each of our collected apps to an API level;
- Trained a classifier on the apps assigned to each level;
- For each level, trained a "paired" classifier on a random selection of apps from all levels; and
- Compared the performance of the level-specific classifiers to the classifiers trained on apps from many levels.

We now discuss these steps in more detail.

Gathering Apps from Google Play

Recall that we began answering our first research question by downloading 1.5 million Android apps from the PlayDrone snapshot of Google Play [109]. To answer our second question, we needed to extract feature vectors from these apps to use as input to our machine learning classifiers. We successfully extracted features from 1,366,317 of these apps; see the discussion of feature extraction below.

To perform supervised machine learning, it is necessary to label each app in the input dataset as benign or malicious. While the majority of Google Play apps are benign, it would be naive to assume that there are no malicious apps in a corpus of 1.3 million. Thus, to assign labels to the apps from Google Play, we retrieved scan reports from the VirusTotal analysis service[5] using the apps' SHA-256 hashes. VirusTotal analyzes each app with a number of antivirus tools and reports which tools flag the app as benign and which flag it

[5] `http://www.virustotal.com`

as malicious. The exact number of antivirus tools used depends on the app and the time at which it was scanned and, for our dataset, ranges from 2 to 58. Of the 1,366,317 apps we analyzed, 71,224 did not have existing scan reports in VirusTotal's database. Due to VirusTotal's public API limits[6], we omit these apps from our corpus instead of submitting them for analysis; as they represent only 5% of the apps, we felt this was an acceptable threat to validity. We also omitted two apps that had scan reports but were scanned by 0 antivirus tools. In total, we were able to label 1,295,091 apps.

As previously mentioned, VirusTotal reports a detection ratio rather than a boolean decision for each app. Figure 2.5 shows the percentage of tools that flagged each app as malicious. The vast majority of apps – 1,105,297 or 85% – were flagged by zero tools. The remainder were flagged by at least one antivirus tool. However, some of these tools will produce false positives, so we did not want to label an app malicious if it was flagged by a small number of tools. Instead, we required that at least 27% of the tools that scanned a particular app flagged it as malicious (recalling that different apps were scanned by a different number of tools). This corresponds to the division between the ninth and tenth vertical bars in Figure 2.5, or where the "hump" that peaks at 25% begins to drop off. Consequently, we labelled 38,336 apps or 3% of the total as malicious and 1,256,755 apps or 97% as benign.

One may wonder if this value reflects the true ratio of malicious to benign apps in Google Play. Unfortunately, it is impossible to know the true ratio, and estimates of it vary widely. Google claims the ratio is very low:

> *With respect to "malicious" applications, less than 1 out of every million installs of an application observed by Verify Apps abused a platform vulnerability in a manner that we think it would be appropriate to characterize as "malicious". [32]*

On the other hand, security researchers claim it may be one in a hundred or higher [77]. We believe 3% is not unreasonable considering that our apps represent more than half the apps in the store.

Gathering Additional Malicious Apps

We supplemented the corpus with malicious apps provided by the Android Malware Genome Project (AMGP) [121], McAfee, and the Drebin project [7]. As these collections are curated

[6] The maximum is four requests per minute and submitted apps have the lowest priority of all scans; consequently, one must wait and repeatedly query for the result.

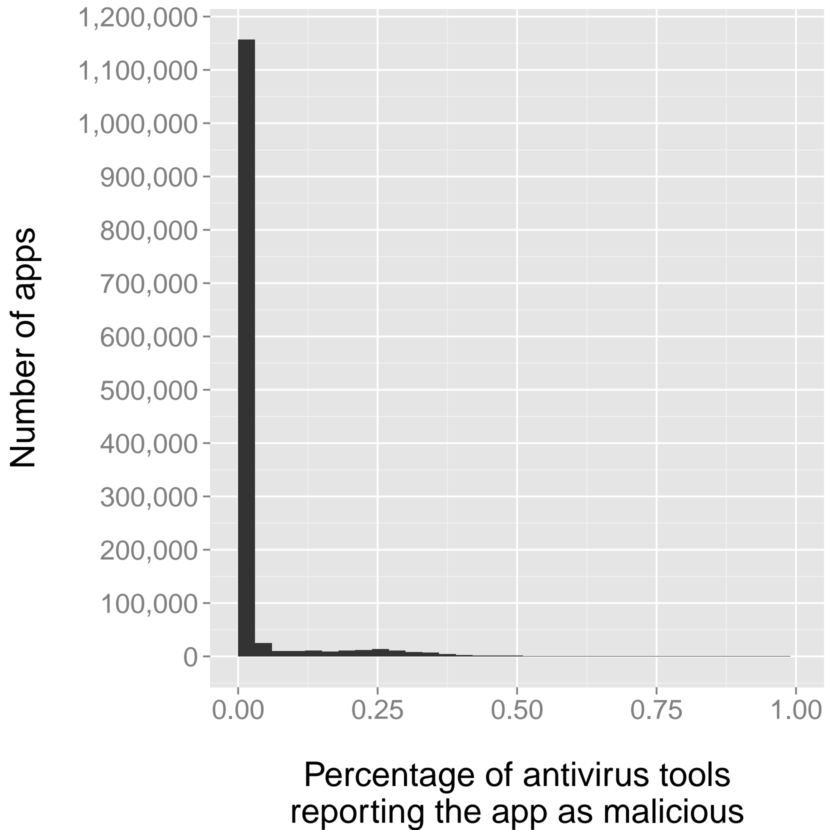


Figure 2.5: The percentage of VirusTotal scanning tools that marked the apps we obtained from the Play store as malicious.

by experts, we assume they are correctly labelled and do not perform any further analysis. These three corpora totalled 7,120 apps, of which we were able to extract features from 7,036; see the discussion of feature extraction below.

To evaluate the detection threshold we selected in the previous step, we submitted these 7,036 apps to VirusTotal. As Figure 2.6 shows, some malicious apps are missed by the service and are flagged by less than three percent of the tools. However, most apps were flagged as malicious by at least 27% of the tools, consistent with our previously selected threshold.

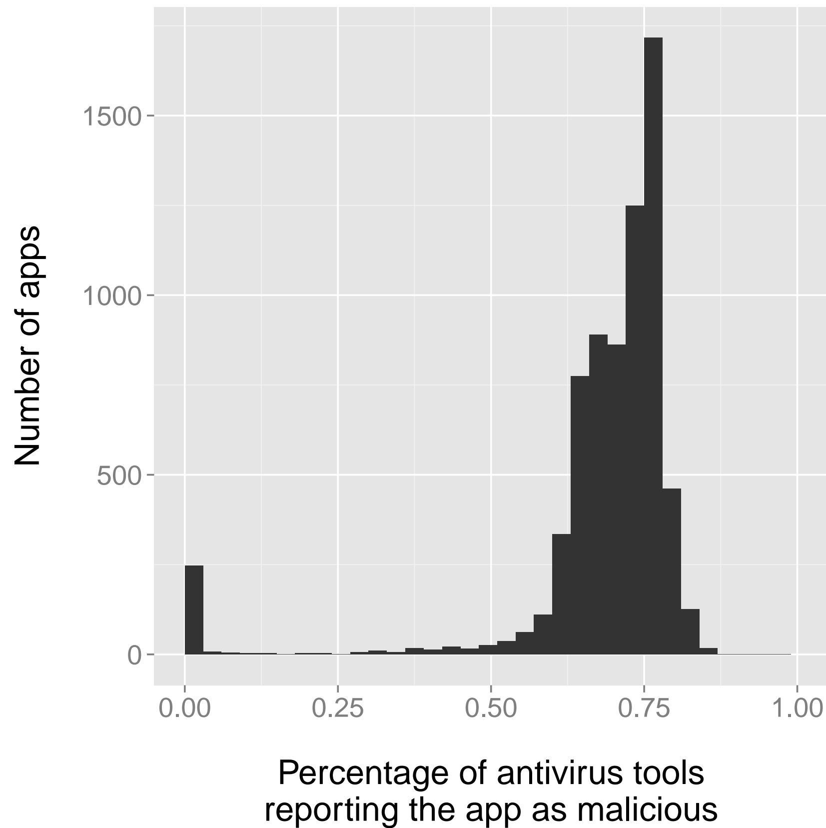


Figure 2.6: The percentage of VirusTotal scanning tools that marked the apps we obtained from malware corpora as malicious.

Assigning Apps to an API Level

In order to determine if the API level is a confounding variable when evaluating a classifier, it was necessary to assign each app to an API level between 1 and 22, inclusive. In theory, it is easy to identify the API level for an app, as developers can indicate an app's target API level in its manifest file.[7] In practice, this information is optional and we found that it was missing for many apps. In addition, even when it is present, the developer can specify a minimum and maximum level rather than a target level, which is not precise enough for our purposes. Finally, the information in the manifest is not guaranteed to be correct. For these reasons, we needed to use a more accurate approach to determine the API level of

[7] `https://developer.android.com/guide/topics/manifest/uses-sdk-element.html`

each app. At a high level, we identified the API level in which each Android method and permission was introduced, identified which Android methods and permissions were used by each app, and chose each app's level based on the levels of the methods and permissions it used. Our procedure for the first two steps was discussed in Section 2.2.1; however, for this part of the study, we also considered the permissions used by the app. To get a list of permissions and the API level in which each was added, we parsed the Android developer documentation.[8] We identified 152 permissions in total in the 22 levels. We parsed the manifest file included in each app to get a list of the declared permissions.

After identifying the Android methods and permissions used by the app, we identified the level that each method and permission mapped to, and selected the maximum of those levels as the app's level. For example, if an app uses methods from levels 1, 4, and 8, we chose level 8 as the app's level. The level therefore represents the most modern features used by the app.

Figures 2.7 and 2.8 show the distribution of apps over levels. Specifically, Figure 2.7 shows the distribution of Google Play apps over the 22 Android levels and whether each app was labelled benign or malicious according to VirusTotal and our selected detection ratio threshold. Two apps from Google Play could not be assigned to a level using our procedure as they did not use any Android API methods; we omit these two from the analysis. In addition, as level 2 of the API does not have any `api-diff` files available, we have no methods mapped to that level. We do have two permissions mapped to level 2, however, so some apps were assigned to that level. Level 6 of the API did not add any methods or permissions, so we have no apps mapped to level 6.

Figure 2.8 shows the distribution of known-malicious apps from the AMGP, Drebin, and McAfee corpora over the 22 Android API levels. The apps from these three sources are graphed in a different figure for three reasons. First, there are 100 times as many apps from Play as from the other three sources; this would make it difficult to see the apps from non-Play sources if the figures were combined. Second, the apps in Figure 2.8 are virtually certain to be malware, since the collections are curated by experts; the labels for the apps in the Play figure are assigned based on a VirusTotal threshold and so are less certain. Third, comparing the two figures shows that apps from Google Play tend to be assigned to newer API levels, which is not surprising given that many app authors update their apps. Apps from the malware corpora are biased toward older API levels, as they were collected some time ago. Table 2.6 provides more information about the number of apps assigned to each level and the percentage of apps in each level that were labelled malicious.

[8] `http://developer.android.com/reference/android/Manifest.permission.html`

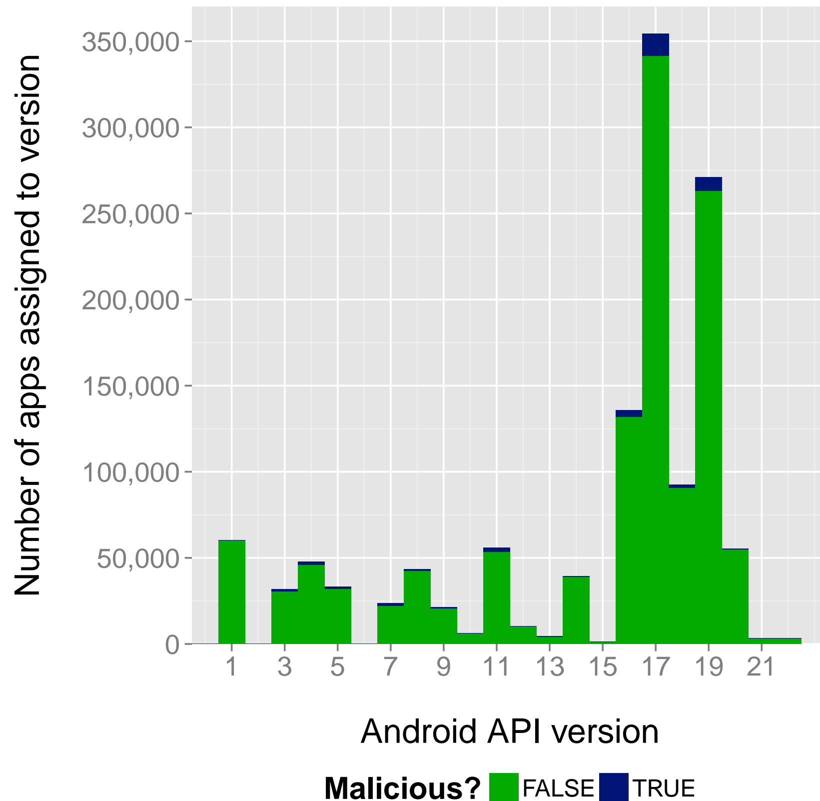


Figure 2.7: The distribution of Google Play apps among Android API levels. Colour indicates whether or not the app was assigned to the malicious category.

Training Classifiers for Each Level

The classifiers we trained for this study varied in three dimensions. First, they varied in *whether app level was controlled*; that is, whether the apps used to train and test the classifier were all assigned to the same API level or not. Second, they varied in the *classification algorithm* used: we tried both k-nearest neighbours and random forest classifiers. Third, they varied in the *feature set* used: we tried both a method-based feature set and a metrics-based feature set. We discuss these three dimensions in more detail below.

Whether App Level Is Controlled The primary aspect we varied for this experiment was whether the app level was controlled. In the fixed-API-level situation, for each API

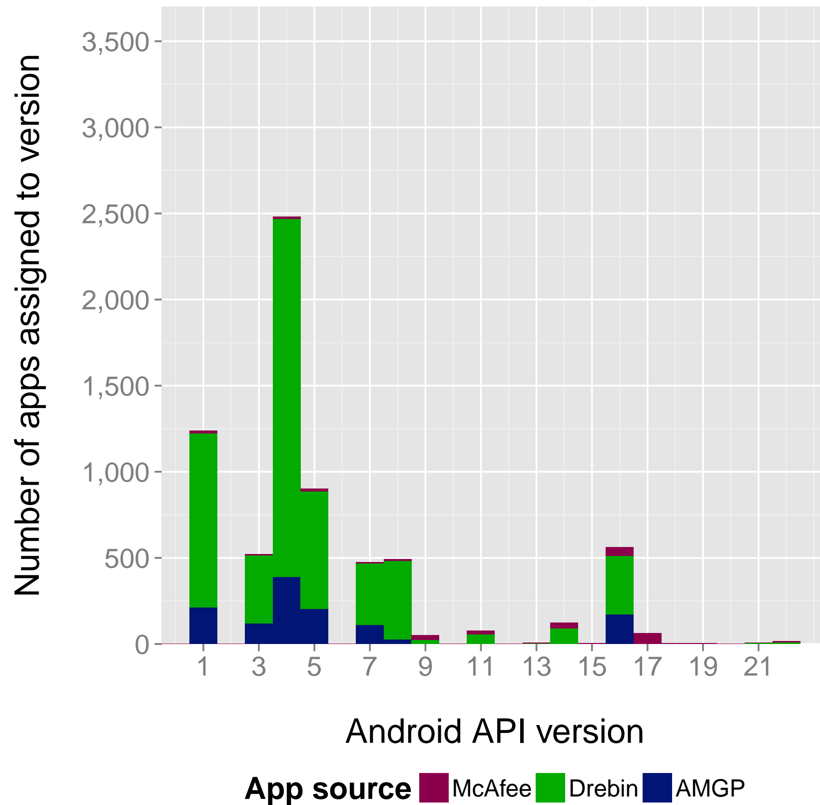


Figure 2.8: The distribution of malicious apps from the Drebin, McAfee, and AMGP corpora among Android API levels.

level, we trained a classifier on the apps assigned to that level. In the random situation, for each API level, we trained a classifier on the same number of apps as were assigned to that level, but selected at random from all API levels. For example, the level 13 classifier was trained and tested on the 4,444 apps assigned to that level; the paired classifier was trained and tested on 4,444 apps randomly selected from all levels. We omitted levels 2 and 6 from the experiment as they did not have enough apps assigned to them to produce meaningful results.

Classification Algorithm We applied two different classifiers: k-nearest neighbours and random forest. The k-nearest neighbour classifier embeds every training vector into a vector space. New apps are classified by embedding them into the same vector space, finding the

Table 2.6: The number of apps from our dataset assigned to each level. The third and fourth columns give the number of malicious apps as a count and as a percentage of the total, respectively. The table includes the 1,295,091 apps from Google Play and the 7,036 from other sources.

API level	Apps assigned to level	Malware assigned to level	Malware (% of total)
1	61,391	1,342	2.2
2	19	2	10.5
3	32,389	1,977	6.1
4	50,177	4,292	8.6
5	34,209	2,081	6.1
6	0	0	0.0
7	24,134	1,941	8.0
8	44,003	1,421	3.2
9	21,421	700	3.3
10	6,361	9	0.1
11	56,091	2,634	4.7
12	10,332	55	0.5
13	4,444	331	7.4
14	39,630	610	1.5
15	1,230	11	0.9
16	136,559	4,600	3.4
17	354,415	13,037	3.7
18	92,570	1,958	2.1
19	270,993	7,847	2.9
20	55,273	482	0.9
21	3,196	17	0.5
22	3,288	25	0.8
All levels	1,302,125	45,372	3.5

nearest k points in the space, and choosing the label that matches the majority of the neighbours. This classifier type has the advantage of making few assumptions about the data. In particular, if different malware families differ from each other as well as from the benign apps, kNN is capable of identifying these "neighbourhoods" in the feature vector space. The main disadvantages are classification time and the so-called "curse of

dimensionality". Classification time will grow in proportion to the number of vectors in the training set unless an approximation technique such as locality sensitive hashing [5] is used. The "curse of dimensionality" refers to the fact that, in high dimensional spaces, the nearest neighbours may no longer be very near, reducing classification accuracy. Nonetheless, kNN is a fairly popular classifier type: it was used by four of the twenty-one studies discussed in Chapter 2.1 [1, 8, 98, 112].

Random forest classifiers are based on decision trees. In a decision tree, each node of the tree contains a rule based on the input features that is used to split the training examples at that node into two or more groups. The leaves of the tree represent classification labels; an app is classified by successively applying rules starting at the root of the tree until a leaf is reached. Decision trees have a tendency to produce very specific rules that overfit the training data. This can be mitigated by training multiple trees: each tree is trained with a random subset of the input data and the final classification is determined by majority vote. A classifier of this type is called a random forest. Again, the classifier makes few assumptions about the input data, and in this case, classification can be done in time proportional to the logarithm of the number of training samples. Random forests were used by nine of the twenty-one studies discussed in Chapter 2.1 [2–4, 30, 64, 92, 100–102].

We used these two approaches to ensure that our results were not due solely to the choice of classifier. For both classifier types, we used a slight variation of ten-fold cross validation for training. We split the data into ten pieces and used eight of them for training. One piece was used to tune the parameters of the classifier – in particular, the value of k or the number of trees in the forest, depending on the classifier type – and the final piece was used for testing. Over the ten folds, every piece was used as the tuning set exactly once and the testing set exactly once. We used scikit-learn [89] for the training.

Feature Set In Section 2.1, we saw that the majority of existing studies that use static analysis to derive classifier features depend explicitly or implicitly on the methods and/or permissions used by an app. However, there were two studies that did not: the first used the strings in the app as features [102], while the second used complexity metrics as features [92]. We sought to understand the impact of controlling for API level in both situations; that is, when methods and permissions are used as features and when they are not. We hypothesized that if classifier accuracy drops when the API level of the apps is controlled, we would see a stronger effect when methods are used as features than when they are not. Consequently, we used two different feature sets. The first feature set is the API method calls and permissions used by the app, determined using Androguard. The 22 levels of the Android API define 21,978 different methods and 152 permissions for a total

of 22,130 features. The second feature set is the set of 144 code complexity metrics used by Protsenko and Müller in their study [92]. We normalized the metrics (subtracted the mean and divided by the standard deviation) to ensure that no one metric would have a disproportionate influence on the classification simply because it has high variance.

We did not include app strings as features because the number of features in this feature set grows with the number of apps analyzed. While this may not have been an issue for the 666 apps used in the original study [102], it would have produced a computationally infeasible number of features for this study.

As mentioned earlier in this chapter, we could not extract feature vectors from all of the apps we collected. Specifically, we obtained vectors for 1,366,317 Google Play apps and 7,036 apps from our three malware corpora. This was due to either Androguard or Soot (on which Protsenko and Müller's metric extractor is based) failing to complete their analyses.

Summary

In total, we obtained a feature vector and, when applicable, a VirusTotal report for 1,302,125 apps. Varying the training parameters described above produced eight configurations per API level. As we omitted levels 2 and 6, we trained classifiers for 20 API levels, giving us a total of 160 classifiers.[9]

2.3.2 Results

In this section, we discuss whether controlling API level reduced classification ability. We also explore how classification ability was affected by the choice of classifier type and feature set. In each case, we compared the paired classifiers (the level-specific and random ones) using the Wilcoxon signed rank test. We chose to compare the F-scores of the classifiers as this score captures both precision and recall and thus simplifies presentation. In any case, we are primarily interested in the relative performance of each pair of classifiers rather than their absolute performance. If, as we hypothesized, API churn has led to overly optimistic evaluations in previous studies, we would expect the API-level-specific classifier to perform worse than the paired random classifier in most cases (i.e., we would expect the blue points to have lower y-values than the green points in Figures 2.9 through 2.12). In practice, we saw the opposite relationship, as we discuss further below.

[9]Of course, as we used ten-fold cross validation, in practice we trained 1,600 classifiers.

kNN, Methods as Features

Figure 2.9 shows the results obtained when we trained kNN classifiers using methods as input features. The figure shows that the classifiers trained and tested on apps from a particular API level almost always had a higher F-score than the ones trained on the same number of randomly selected apps. Using R's Wilcoxon signed rank test indicates that the difference between the means was significant with a p-value of 0.0007. The effect size was also large: using R's function for Cohen's d reveals an effect size of 0.98.

Careful examination of the figure will reveal that two data points are missing: the results for the level-specific classifiers for levels 10 and 15. In these two cases, the classifier never predicted a positive (malicious) label. Recalling that precision is the number of true positives divided by the sum of the true and false positives, it is clear that precision in this case is undefined, and consequently, the F-score is also undefined. The paired points, or the random classifiers for levels 10 and 15, were omitted when computing the p-value and effect size as those calculations require pairs of measurements.

kNN, Metrics as Features

Figure 2.10 shows the results obtained when we trained kNN classifiers using metrics as input features. As in the previous scenario, the classifiers trained and tested on apps from a particular API level almost always had a higher F-score than the ones trained on the same number of randomly selected apps. Using R's Wilcoxon signed rank test indicates that the difference between the means was significant with a p-value of 7.6e-05. The effect size was also large: using R's function for Cohen's d reveals an effect size of 1.09.

As in the previous scenario, the level-specific classifiers for levels 10 and 15 never predicted the label "malicious" for the apps in the test set in any of the ten folds. Thus, the paired points were omitted from the p-value and effect size computations.

Random Forest, Methods as Features

Figure 2.11 shows the results obtained when we trained random forest classifiers using methods as input features. As in the previous two scenarios, the classifiers trained and tested on apps from a particular API level almost always had a higher F-score than the ones trained on the same number of randomly selected apps. Using R's Wilcoxon signed rank test indicates that the difference between the means was significant with a p-value of 0.0003. The effect size was also large: using R's function for Cohen's d reveals an effect size of 1.04.

Random Forest, Metrics as Features

Figure 2.12 shows the results obtained when we trained random forest classifiers using metrics as input features. As in all the other scenarios, the classifiers trained and tested on apps from a particular API level almost always had a higher F-score than the ones trained on the same number of randomly selected apps. Using R's Wilcoxon signed rank test indicates that the difference between the means was significant with a p-value of 3.6e-05. The effect size was also large: using R's function for Cohen's d reveals an effect size of 1.24.

Answer 2.2. *API level does not need to be controlled when evaluating classifier accuracy.*

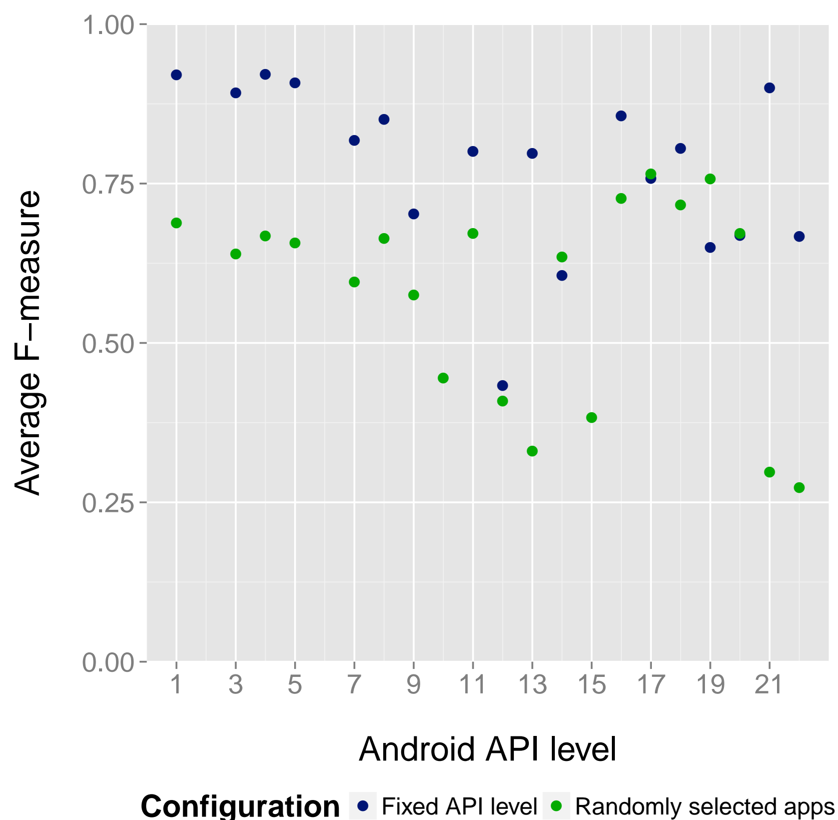


Figure 2.9: The F-scores of k-nearest neighbour classifiers that used methods as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis.

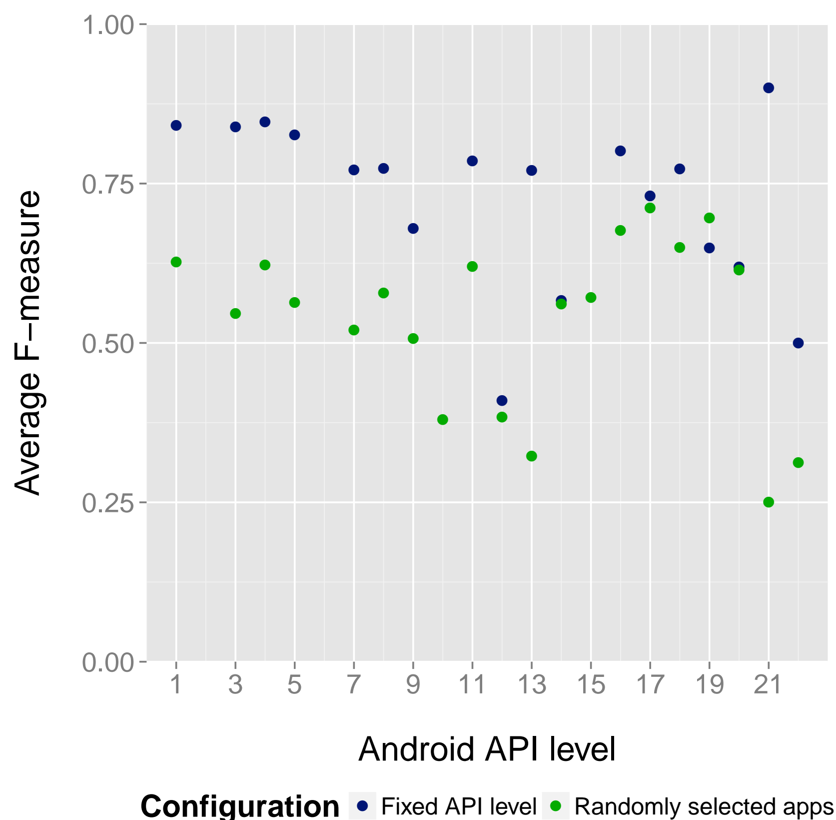


Figure 2.10: The F-scores of k-nearest neighbour classifiers that used metrics as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis.

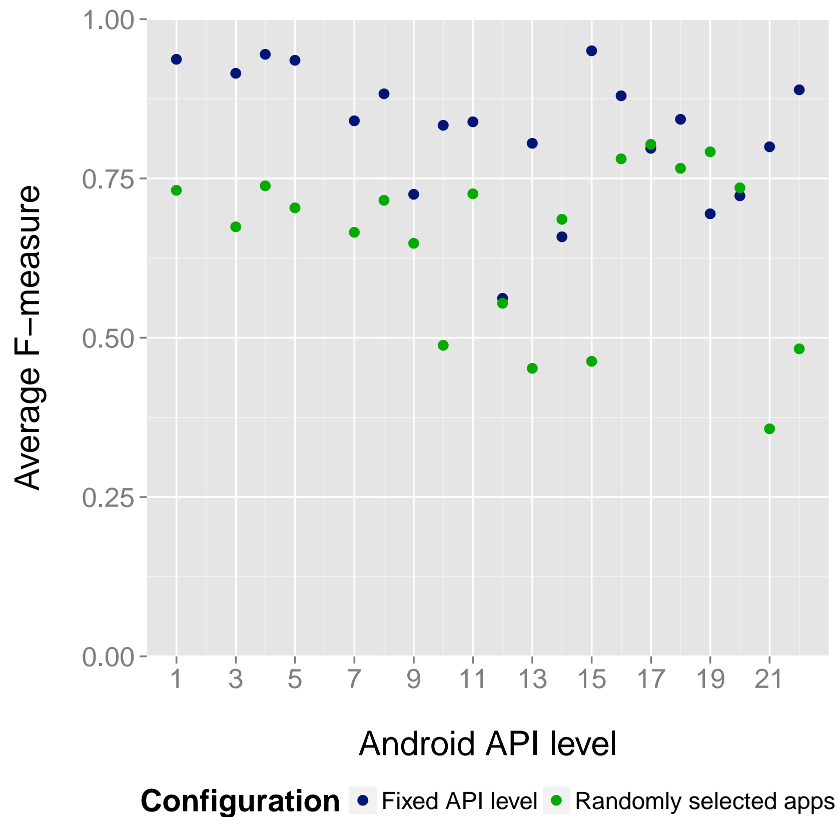


Figure 2.11: The F-scores of random forest classifiers that used methods as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis.

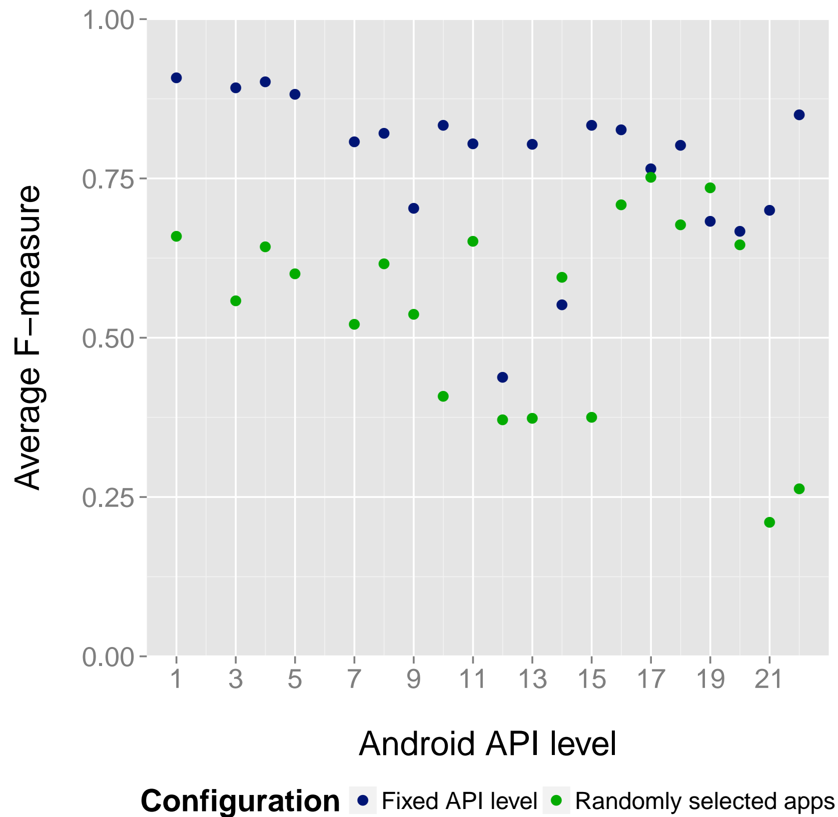


Figure 2.12: The F-scores of random forest classifiers that used metrics as features. As we used ten-fold cross validation, each point represents the average f-score of the ten folds. The x-axis shows the Android API level that the apps belong to for the level-controlled classifiers (blue points). The paired classifiers (green points) are trained and tested using the same number of apps as the level-controlled classifier at the same position on the x axis.

2.4 Discussion

We begin the discussion of our results by answering the research questions defined in the introduction to this chapter. With regards to our first research question, it seems that 1,000 apps may be sufficient for studies that consider "normal" behaviour, or things that the majority of apps do. Detecting outliers naturally requires more apps, and over a million may be required for a generalizable study.

The results from our second research question provide a way of testing this guideline. Recall that for each API level, we generated a paired classifier that contained a random selection of apps. These corpora contained different numbers of apps, allowing us to see how classifier accuracy changes with corpus size. Figure 2.13 shows this information; we can see that the accuracy of the classifiers increases with size and is still increasing for the largest corpus we considered (354,415 apps). This provides additional data to support our claim that at least 1,000,000 apps may be needed to detect unusual behaviour.

With regards to our second research question, we found that API level does not need to be controlled when evaluating classifier accuracy. In fact, when API level was controlled, classification accuracy actually *increased*, regardless of the feature set used or the classifier type. This indicates that the existing work discussed in Section 2.1 is not affected by this potential threat to validity.

An open question is the cause of this increase in accuracy. Lamba et al. [62] studied how Android app developers use the API and found that they frequently call the same groups of methods together in their own user-defined methods. These API usage patterns offer a possible explanation for our results. Apps that use the same API level are likely to use many of the same patterns, which may help the classifier distinguish malware from benign apps. However, when training a classifier on apps from many levels, the patterns are different for each level and so are less useful for classification. We plan to explore this possibility by re-analyzing the callgraphs from the apps in our study. We also plan to measure the information gain of the input features to see which ones are the most useful for classification; this may help explain our findings further.

Another possible use of our results is to help Android API maintainers make maintenance decisions. APIs require a great deal of maintenance [39], but this maintenance is often done in an ad-hoc way, without collecting data about how the API is used by clients [120]. Without such data, it is difficult to make informed decisions about the evolution of the API. Changes that seem reasonable to the API maintainers may end up having a significant negative effect on API clients.

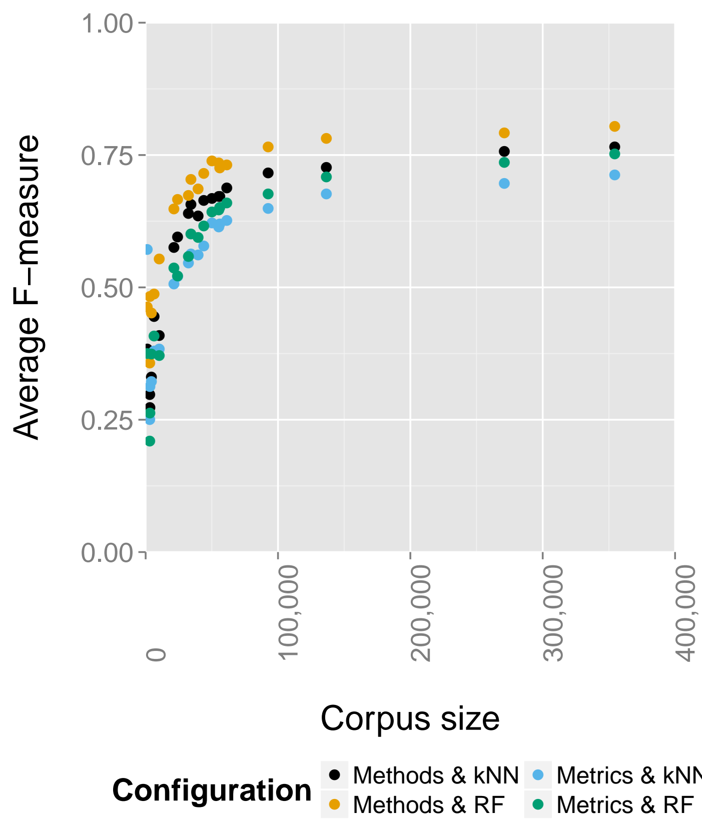


Figure 2.13: The F-scores of the paired classifiers (randomly selected apps) as a function of the number of apps used to train and test the classifier.

The mailing list archives of the Seaside project provide an example of this. First, an API user asked about several classes that he could not find in the latest version of the framework:

> *In looking at Seaside 2.81.390 I noticed that the Seaside 2.6 dialog classes listed below are not in Seaside 2.8a1.390. I have not been paying attention to Seaside 2.8 so don't know much about its status. I am wondering if these classes have been dropped, have not been ported to 2.8 or does their functionality exists elsewhere?*
>
> *Missing Dialog classes*
> *WAChangePassword*
> *WAEditDialog*
> *WAEmailConfirmation*
> *WAGridDialog*
> *WALoginDialog*
> *WANoteDialog*

To which one of the Seaside developers replied:

> *They have been dropped. A mail went out to this list if anybody still used them and nobody replied. [111]*

Changes that negatively impact the API users in this way harm their opinion of the API and, in some cases, may lead them to adopt a different API. The problem of negatively impacting clients is particularly relevant for the Android framework, as changes to the API impact both the app developers and the customers using the apps. Unnecessary, burdensome changes may drive away app developers, while apps with high code churn have been shown to have lower user ratings [35]. Thus, the API maintainers could benefit from the use of data science to determine the most and least popular API methods.

Concretely, we found that 4,547 of the 21,971 Android API methods we identified were not used by any of the apps in our largest corpus (1,368,376 apps). However, 1,050 of these methods belong to abstract classes, 519 belong to interfaces, and 17 of the methods in concrete classes were removed. If we conservatively assume that a static analysis could not detect uses of the methods in the abstract classes and interfaces, this still leaves 2,961 methods that are not used by the apps in our corpus. In other words, a non-trivial portion

(13%) of the Android API does not appear to be providing any benefit to the apps we analyzed. Given the large size of our corpus, it seems safe to say that these methods contribute little to the API. Removing them could shrink the API, reducing maintenance effort for the Android team while also reducing the number of API methods that API users must consider when searching for the functionality they desire.

In fact, the API developers themselves are sometimes aware that the methods contribute little to the API. For example, both versions of the method `isBoring` in the `android.text.BoringLayout` class are not used by the apps we studied; the documentation for this class states: "A BoringLayout is a very simple Layout implementation for text that fits on a single line and is all left-to-right characters. You will probably never want to make one of these yourself."

To see some other candidates for deprecation, we list the ten least popular methods in our dataset in Table 2.7. None of these methods were used by the apps in our corpus. Three of the methods belong to abstract classes: `ClipboardManager.init` (a constructor), `getPackageInstaller`, and `addNodeChangeListener`. They may have zero recorded calls because the static analysis was not capable of detecting calls to concrete implementations. The method `stopLeScan` was deprecated in level 21, suggesting that the API maintainers are aware that it is not often used. However, the remaining six methods provide examples of methods that have not been deprecated and do not belong to abstract classes or interfaces, but are not used by app developers. These methods may therefore be logical candidates for deprecation.

The four studies of Android API method usage discussed in Section 2.1 considered how developers use Android API methods, as we do, but did not consider the *least* popular Android methods with an eye to deprecation. In addition, two used different populations of apps and methods: Li et al. [65] considered only malware, while Li et al. [66] considered only "inaccessible" API methods. Thus, our results provide a new perspective on how mining method usage from a large corpus can be used to guide deprecation.

2.5 Threats to Validity

As our work relies on Androguard, a static analysis tool, the usual static analysis shortcomings apply. If an app leveraged reflection to access an API method, we would not identify this call. More importantly, if an app leveraged an API by extending a class, we would also exclude such a call, unless it was a direct call (such as `super()`) to a call in the Android API.

Table 2.7: The ten least popular methods from the Android API in our dataset. Each method descriptor has the format `className::returnType::methodName::(methodArguments)`. None of the methods were called by apps in our dataset.

Rank	Method descriptor
21,962	`java.util.BitSet::BitSet::valueOf::(long[])`
21,963	`android.view.InputDevice::int::getControllerNumber::()`
21,964	`android.text.ClipboardManager::ClipboardManager::init::()`
21,965	`android.content.pm.ServiceInfo::int::describeContents::()`
21,966	`java.util.LinkedList::E::removeLast::()`
21,967	`android.bluetooth.BluetoothAdapter::void::stopLeScan:: (android.bluetooth.BluetoothAdapter.LeScanCallback)`
21,968	`android.test.mock.MockPackageManager::List:: getPreferredPackages::(int)`
21,969	`android.content.pm.PackageManager::PackageInstaller:: getPackageInstaller::()`
21,970	`java.util.prefs.AbstractPreferences::void::addNodeChangeListener:: (java.util.prefs.NodeChangeListener)`
21,971	`java.util.concurrent.Executors::Callable::callable:: (java.lang.Runnable, T)`

An additional threat to validity is whether the apps we used are a representative sample of the universe of apps. We attempted to mitigate this threat in several ways. First, we used a large number of apps from the PlayDrone corpus: over half the apps that are available from the Play store. Second, we used three different malware corpora: AMGP, Drebin, and apps from McAfee. Each corpus might be biased towards a particular type of malicious app (e.g., command and control apps) or a particular age of app, but by using all three corpora, we have increased the variety of malware in our corpus. Thus, we feel these apps are representative of the apps available for download from the Play store.

Unfortunately, using apps from the Play store introduces another threat to the validity of our second study: whether or not the apps are correctly labelled as benign or malicious. We attempted to mitigate this threat by choosing a detection threshold based on the VirusTotal distribution histogram and ensuring that the percentage of apps labelled malicious was in line with existing estimates. This is not a perfect solution, but as it was infeasible to manually examine each of the 1.3 million apps, we feel it is an acceptable threat.

Another threat to the validity of our malware detection work is selection bias. Google employs the Bouncer tool [81] to detect and remove some malware from the Play store. Consequently, the malicious apps we obtained from Play via PlayDrone may not be a representative sample of all malicious Android apps. However, they *are* a representative sample of malicious apps that were not detected by existing malware detection tools, which may be an even more interesting sample. In addition, we removed the apps that did not have existing VirusTotal reports for efficiency reasons. While these represent a small percentage of the apps, it may have biased the results.

A final threat to the validity of our second study is spuriousness; that is, the possibility that the relationship between API level and classification accuracy we observed is due to some other, third variable. We attempted to mitigate this threat by trying two different feature sets and two different classifier types to ensure that the same relationship was observed in different situations. We also note that, although classifier accuracy depends on the number of apps used to train the classifier, as illustrated in Figure 2.13, this is not a threat to our work. This is because the paired classifier for each level uses the same number of apps as the level-specific classifier, so the size of the corpus is controlled for. That is, we are interested in the relative accuracy of each pair of classifiers rather than their absolute accuracy.

2.6 Relevance to Thesis

Recall that our thesis statement is as follows:

Thesis Statement. *Robust development processes are necessary to minimize the number of faults introduced when evolving complex software systems. These processes should be based on empirical research findings. Data science techniques allow software engineering researchers to develop research insights that may be difficult or impossible to obtain with other research methodologies. These research insights support the creation of development processes. Thus, data science techniques support the creation of empirically-based development processes.*

App development processes depend on the work done by software engineering researchers who study mobile apps. Until now, there have not been any empirical guidelines suggesting the number of apps researchers should use in these studies. Some studies use as few as ten apps (e.g., [73]). This has made it difficult to judge the generalizability of any particular study and thus whether the results are useful to app developers. Our guidelines can help improve app research practices, which in turn will improve app development processes.

Mobile malware detection tools likewise rely on research findings to determine, for instance, which classifier type will perform the best and which input features should be used. If prior work had been based on an incorrect assumption, namely, that API level is irrelevant, their results would be invalid and thus of little use to developers creating malware detection tools. As it happens, API level does not need to be controlled to obtain correct results. This information gives us more confidence in the findings of previous studies and thus in the application of their results to the development of practical tools for detecting malicious apps.

The use of data science was critical to obtain our results. Studying 1.3 million apps and 11 TB of callgraphs would not have been possible without the use of computing clusters, big data storage and handling techniques, statistics, machine learning, and various other tools that belong to the data science toolkit. Thus, this work provides an example of how data science techniques can be applied to develop new research insights that in turn assist software developers.

Chapter 3

Measuring Test Suite Costs and Benefits

> *"But what's the harm in over-testing, Phil, don't you want your code to be safe? If we catch just one bug from entering production, isn't it worth it?" [...] This line of argument is how we got the TSA. [36]*

Buggy software costs its developers money. Among other things, bugs can discourage new customers from adopting a product and can drive away existing customers. Consequently, many techniques exist for avoiding the introduction of bugs and for quickly identifying and fixing bugs when they are introduced anyway. However, these techniques come with costs of their own, so developers must carefully assess their cost-effectiveness before deciding whether and to what extent to adopt them.

Automated regression testing [33] is one technique for fault detection that has seen wide adoption [23, 41]. However, it is a costly technique: developers must write, maintain, and regularly execute the test suite to see the benefits of regression testing. Kasurinen et al. [56] conducted a survey of industrial developers that, among other findings, identified development expenses and maintenance costs as the main obstacles to adopting automated testing. One of their participants stated:

> *Developing that kind of test automation system is almost as huge an effort as building the actual project.*

Moreover, one company that experimented with automated testing eventually removed the test suite due to the cost of maintaining it.

As this company's experience indicates, it is important to understand both the costs and benefits of regression testing when deciding whether and how to adopt it. Costs, in this case, include the cost of initially writing the tests, maintaining the test suite, executing the suite, and examining the test output. Benefits include the number of faults found as a result of investing in the test suite. Previous studies have considered the cost-effectiveness of automated regression testing, as we will discuss further in Section 3.1. However, these studies share a common limitation: they were conducted by mining historical data from the test suite repositories. While tracking the evolution of the test suites can provide understanding of how the suites were developed and what their likely maintenance costs might have been, they do not permit measuring possible benefits in terms of detected faults.

To address this limitation, we studied 61 projects that use Travis CI[1], a cloud-based continuous integration tool. Travis builds a project and executes its tests every time a developer pushes a change or opens a GitHub pull request, meaning that developers push their commits to the repository *before* testing them, frequently introducing faults that cause one or more tests to fail. As Travis integrates with project version control systems, when test failures occur, it is possible to precisely determine whether changes to the tests, the system under test, or both were required to make the test suite pass again. Once flaky tests[2] are accounted for, if a test failure is resolved by changing the source code, we know that the test provided a benefit by detecting a fault. On the other hand, if the failure is resolved by changing the test itself, we can conclude that the test was buggy or obsolete; in other words, we can conclude that the change represents a maintenance cost. Examining the Travis results therefore allows us to measure both the costs and benefits of regression testing, in contrast to previous work. Specifically, we considered the following research questions:

Research Question 3.1. *What proportion of test-suite executions are flaky?*

Research Question 3.2. *Once these flaky test-suite executions are accounted for, what proportion of test-suite failures represent a maintenance cost and what proportion represent a benefit?*

Research Question 3.3. *Why do tests usually require maintenance, and can maintenance costs be reduced?*

[1] `https://travis-ci.org/`

[2] Tests that pass and fail non-deterministically.

By studying a dataset of 106,738 Travis builds, which we describe in Section 3.2, we found that the benefits of regression testing are lower than one might expect relative to the costs. Briefly, we found that 18% of test suite executions fail and that 12.8% of these test suite failures are due to flaky (non-deterministic) tests. Of the non-flaky test suite failures, only 74.1% are caused by a defect in the system under test; the remaining 25.9% are due to tests that are either incorrect or obsolete. The causes of test maintenance vary widely, and some of the causes are avoidable. Section 3.4 provides full answers to the three research questions given above.

Although we feel these findings provide valuable information to developers in themselves, they can also be used to inform test case selection techniques. The goal of test case selection is to reduce the cost of running the regression suite by running only a subset of tests each time. Ideally, one would only select and execute the tests that are going to fail and detect a fault; of course, in practice, one does not know *a priori* which tests will do so. Our dataset of Travis builds allowed us to address an additional research question related to test selection:

Research Question 3.4. *What proportion of test executions expose defects?*

In brief, we found that, in failing builds, an individual test execution has only a 0.28% chance of failing and exposing a real defect for any given code change. Section 3.5 provides a full answer to this research question.

Following the presentation of results, Section 3.6 describes threats to validity and Section 3.7 describes our data replication package. Section 3.8 explains how the work in this chapter supports our thesis statement.

3.1 Related Work

As mentioned above, previous work has considered the cost/benefit tradeoff of automated regression testing. We first discuss studies that used repository mining to measure the cost of maintaining a regression test suite. Next, we discuss studies that captured test outputs, but were not able to use this information to measure the costs and benefits of regression testing. Finally, we briefly discuss studies that attempted to decrease the costs of regression testing by reducing the amount of time required to execute the suites. These particular studies of test selection and prioritization are closely related to our own work.

3.1.1 Measuring Test Suite Maintenance Effort

Seven previous studies have considered test suite maintenance costs by exploring how test suites evolve over time. Zaidman et al. [117] examined how developers evolve their test suites along with the test code and whether testing effort varied by the project schedule. They found both close synchronous evolution as well as separate, stepwise evolution but failed to find any increase in testing effort before a major release.

Pinto et al. [90] studied test evolution and found that 30% of the changes made to test suites were modifications, 15% were test additions, and 56% were test deletions.

Marsavina et al. [70] studied the co-evolution of production and test code. They found that test code changes made up between 6% and 47% of all code changes for the projects they studied.

Beller and Zaidman [13] found that tests and production code have some tendency to change together, but that tests were not changed every time the system under test was changed and vice versa.

Kasurinen et al. [56] conducted a survey of industrial developers that, among other findings, identified development expenses and maintenance costs as the main obstacles to adopting automated testing. In fact, one company that experimented with automated testing eventually removed the test suite due to the cost of maintaining it. This indicates that maintenance is perceived as very expensive, and in at least one case, the cost was high enough that developers could not justify using automated regression testing.

Grechanik et al. [34] estimated that the costs associated with maintaining and evolving test scripts are $50 million to $120 million per year. They also showed that even simple changes could result in 30% to 70% of test scripts needing maintenance, a process that could take hours or days and caused interruption to continuous integration systems. While developers in the study saw the benefits of the automated test suite, the maintenance burden often caused them to throw away their tests and start from scratch, often with faulty logic due to time pressures.

Herzig et al. [41] developed a test selection tool called THEO that selects tests to be executed if the probable cost of executing the test is lower than the probable cost of skipping the test. To calculate these costs, THEO uses the past defect detection rate for each test case (i.e., the true positive rate) and the past false alarm rate of a test case (i.e., the false positive rate). These probabilities can then be used along with data such as machine costs, number of engineers, and inspection costs in order to estimate the cost of skipping or executing a test. An evaluation of the system using historical data from Microsoft projects allowed 35% to 50% of tests to be skipped while only letting 0.2% to 13% of defects escape.

In Section 3.5 we show that, in our dataset, a perfect test selection technique would execute less than 0.38% of tests.

All seven of these studies share the same limitation: they could not capture the execution results of the test suites due to being performed retroactively by mining historical data. This means we cannot measure the benefit provided by the test suites in terms of the number of faults detected. Without this information, we do not have a complete picture of the cost/benefit tradeoff.

3.1.2 Capturing Test Outputs

Four previous studies have managed to capture test outputs. Anderson et al. [46] provide insight into Microsoft Dynamix AX R2, a large industrial project. This project includes over 5.5 million lines of product source code and over 4.8 million lines of regression test code. The authors report that running the regression suite resulted in a 9% test case failure rate during each execution of the test suite.

Memon and Soffa [74] found that 74% of all tests failed between successive releases of a single industrial product, suggesting that test failures are relatively common. Beller et al. [13] report that 65% of IDE-based JUnit test runs fail.

Vasilescu et al. [108] explored the relation between Travis CI build results (success or failure) and the way the build was started (i.e., direct commit from a developer with write access to the repository or a pull request). The authors found that builds started by pull requests are more likely to fail than those started by direct commits. They also found that, although 92% of the projects they studied are configured to use Travis CI, only 42% actually do; we observed the same behaviour, as will be described in Section 3.3.

Unfortunately, while these studies captured test execution results, it is not clear what proportion of the observed failures occurred because of faults and what proportion occurred because the tests required maintenance. Additionally, developers in these studies may have been using test driven development methodologies which would further influence the test failure rates. It is therefore hard to draw conclusions about the amount of developer time devoted to test maintenance and thus the cost/benefit ratio of regression testing from these studies.

3.1.3 Reducing Suite Execution Time

A number of studies have attempted to reduce the cost of regression testing by reducing the cost of *executing* the suite. The basic assumption is that re-running all the tests every

time is too expensive, and some way of reducing this cost is required. As an example, Rosenblum and Weyuker [94] proposed the use of test coverage information to predict the cost-effectiveness of regression testing strategies.

Elbaum et al. [23] used test suite selection at the pre-submit stage and test suite prioritization at the post-submit stage to increase the cost-effectiveness of testing. Note that Google's test suites are large enough that the authors were working with test *suites*, not test *cases*. During the pre-submit phase – i.e., before code was pushed to the central repository – suites that failed during a pre-determined failure window were selected to be executed, the basic intuition being that recent failures are likely to predict future failures. New tests and tests that had been skipped more than a set number of times were also selected during this phase. During the post-submit testing phase, the authors attempted to avoid skipping test suites and thus prioritized rather than selected test suites using the same criteria used during the pre-submit phase. This strategy ensured that all test suites were executed during the post-submit phase while running the suites that were most likely to fail first, thereby shortening the feedback loop.

Anderson et al. [46] developed two test prioritization techniques that used test result history. The first technique, *most common failures*, is based on the intuition that that tests that failed the most in the past are the most likely to fail in the future. The second technique, *failures by association*, attempts to use association rule mining to improve on the first technique. The authors found that the techniques had similar performance when predicting future failures and that both worked better when a small window of recent executions was used, rather than the entire historical dataset.

Most test suite reduction techniques use coverage to detect redundant tests. This leads to a loss in test suite fault detection ability because fully overlapping coverage does not necessarily mean the tests will always fail under the same circumstances [60]. To address this issue, Koochakzadeh and Garousi [59] developed a test reduction tool that brings a human tester into the loop. Their tool identifies potentially redundant tests using coverage analysis, then lets testers inspect these tests to identify the true positives that can be removed from the suite. Using this technique they were able to remove eleven of the 82 tests in their subject system without affecting the suite's mutation score. In contrast, the coverage based approach identified 52 tests as redundant and reduced the mutation score by 31%.

Rothermel et al. [96] studied how the granularity of test suites influenced the cost-effectiveness of regression testing. They concluded that typical regression testing techniques usually do not lose fault detection capability when operating on coarse-grained test suites, but they do tend to save test execution time.

While the information produced by these studies sheds light on the costs of regression testing, they are concerned only with the cost of *executing* the test suites, while we focus on the cost of *maintaining* the suites. In addition, like other studies that we have discussed, the ones above do not measure the benefits of regression testing in terms of faults detected. However, these studies are relevant to our final research question, which provides information that may help researchers interested in test case prioritization and selection.

3.2 Method

The goal of this work was to study the costs and benefits of automated regression testing. To do so, we used the following high level procedure:

1. Identified open source Java projects that use Travis CI and extracted the results of the Travis builds.
2. Identified the status of each build to identify test suite failures.
3. Identified how builds transition between different statuses.
4. Mapped each build to associated code changes in the version control repository.
5. Determined whether each failing build was resolved by changes to the source code, the test code, or both.

We now discuss each of these steps in turn.

3.3 Collecting Travis Build Data

To study Travis builds, we identified a large set of mature Java-based projects that use the Travis infrastructure to actively execute their tests in the cloud. We selected subject programs by querying the GitHub Archive[3] for Java projects that received more than 1,000 push events between 2012 and 2014; this time frame was chosen based on Travis support for Java, which began in February 2012. The query returned 685 projects; of these, 421 projects had Travis accounts. We were able to successfully clone 402 of these projects from GitHub. As the focus of our analysis was on regression testing, we eliminated early-stage projects that were less than two years old. This resulted in 362 projects.

Of these, only 101 projects actively used Travis to execute their test suites; the other 261 projects had signed up for the service but did not use it. Unfortunately, several of

[3] https://www.githubarchive.org/

these remaining projects did not configure Travis correctly or did not examine the Travis output, resulting in long stretches of broken builds; others almost never experienced a failure. To account for this, we removed the 20% of the projects that had the most errors and failures and the 20% that had the fewest errors and failures, resulting in a final set of 61 projects. Table 3.1 lists these projects along with, for each project, the number of source lines of code (SLOC) in the system under test (SUT) and the test suite.[4] Note that we included all languages measured by `cloc`, including XML. Some data points may therefore be surprising; for example, `picketlink` contains over 200,000 lines of XML in the test suite, while `thredds` contains a million lines of XML in the SUT.

[4]Calculated using non-comment source lines using `cloc`: `https://github.com/AlDanial/cloc`.

Table 3.1: The 61 projects considered in this study. The second column lists the number of non-blank, non-comment lines of source code (SLOC) in the system under test (SUT). The third column lists the number of non-blank, non-comment lines of source code in the system's test suite. The fourth column sums these quantities and the fifth column lists the amount of test code in the project as a percentage of the total.

Project	SUT SLOC	Test SLOC	Total SLOC	Test %
picketlink/picketlink	89,667	255,032	344,699	74
SonarSource/sonar-java	109,558	281,201	390,759	72
FasterXML/jackson-databind	59,441	98,735	158,176	62
camunda/camunda-bpm-platform	204,976	330,969	535,945	62
square/okhttp	21,182	31,326	52,508	60
mongodb/mongo-java-driver	64,598	90,678	155,276	58
cloudfoundry/uaa	60,862	77,917	138,779	56
timmolter/XChange	101,633	120,350	221,983	54
cucumber/cucumber-jvm	14,986	16,475	31,461	52
spring-projects/spring-boot	123,828	113,991	237,819	48
apache/jackrabbit-oak	212,822	188,941	401,763	47
weld/core	89,971	79,863	169,834	47
ratpack/ratpack	37,117	31,245	68,362	46
orientechnologies/orientdb	241,898	203,156	445,054	46
SonarSource/sonarqube	316,592	265,168	581,760	46
square/picasso	5,429	4,321	9,750	44
killbill/killbill	96,227	69,565	165,792	42
dynjs/dynjs	34,651	23,440	58,091	40
apache/cloudstack	528,612	310,220	838,832	37
openmrs/openmrs-core	96,001	55,476	151,477	37
yegor256/rultor	9,367	5,107	14,474	35
grails/grails-core	81,843	43,203	125,046	35
netty/netty	167,693	65,935	233,628	28
nutzam/nutz	63,075	23,608	86,683	27

puniverse/quasar	45,517	14,194	59,711	24
ansell/owlapi	123,767	37,316	161,083	23
restlet/restlet-framework-java	118,246	34,561	152,807	23
Jasig/cas	95,438	24,590	120,028	20
lucmoreau/ProvToolbox	43,275	10,888	54,163	20
bndtools/bnd	178,993	35,471	214,464	17
structr/structr	192,159	38,067	230,226	17
Jasig/uPortal	190,901	32,760	223,661	15
CloudifySource/cloudify	74,153	12,629	86,782	15
nodebox/nodebox	28,677	4,649	33,326	14
sorcersoft/sorcer	86156	13,004	99,160	13
anathema/anathema	86,593	12,990	99,583	13
wordpress-mobile/WordPress-Android	159,564	21,890	181,454	12
apache/pdfbox	121,375	16,602	137,977	12
neophob/PixelController	11,832	1,592	13,424	12
SpoutDev/Spout	61,903	7,988	69,891	11
bitsofproof/supernode	9,634	1,148	10,782	11
dcoraboeuf/ontrack	57,541	6,805	64,346	11
SeqWare/seqware	110,023	12,398	122,421	10
orbeon/orbeon-forms	601,787	65,222	667,009	10
Hidendra/LWC	11,034	991	12,025	8
FAP-Team/Fap-Module	115,515	7,768	123,283	6
Unidata/thredds	1,640,653	87,979	1,728,632	5
threerings/tripleplay	19,914	1,056	20,970	5
owncloud/android	65,757	2,940	68,697	4
essentials/Essentials	43,304	1,763	45,067	4
openmicroscopy/bioformats	206,668	6,027	212,695	3
BaseXdb/basex	132,167	3,526	135,693	3
bndtools/bndtools	53,108	729	53,837	1
Graylog2/graylog2-server	161,011	2,180	163,191	1
SpoutDev/Vanilla	62,699	840	63,539	1

RoyalDev/RoyalCommands	25,729	192	25,921	1
BuildCraft/BuildCraft	141,482	816	142,298	1
abarisain/dmix	32,301	186	32,487	1
jOOQ/jOOQ	522,269	1,491	523,760	0
gwtbootstrap/gwt-bootstrap	21,646	45	21,691	0
openmicroscopy/openmicroscopy	685,237	1,230	686,467	0

We examined the total size and churn in the test suites and production code of these systems to estimate how large the testing effort was. The aggregate results for the size of the production code and test code are shown in Table 3.2. This table shows that although test code makes up far less than half of the code overall, it grew faster than production code: the proportion of test code grew from 10.8% to 21.9% (see Figure 3.1). In total, the test code for our systems totalled over 2.9 million SLOC. Even though SLOC may not be a perfect measure of development effort, the results still show that a considerable amount of effort was spent creating and evolving the test suites for these projects.

Table 3.2: Final production and test code size and increase in size over the two year study period.

	Code	Test
Lines of Code	10,479,380	2,942,473
Percentage Increase	59.1	267.8

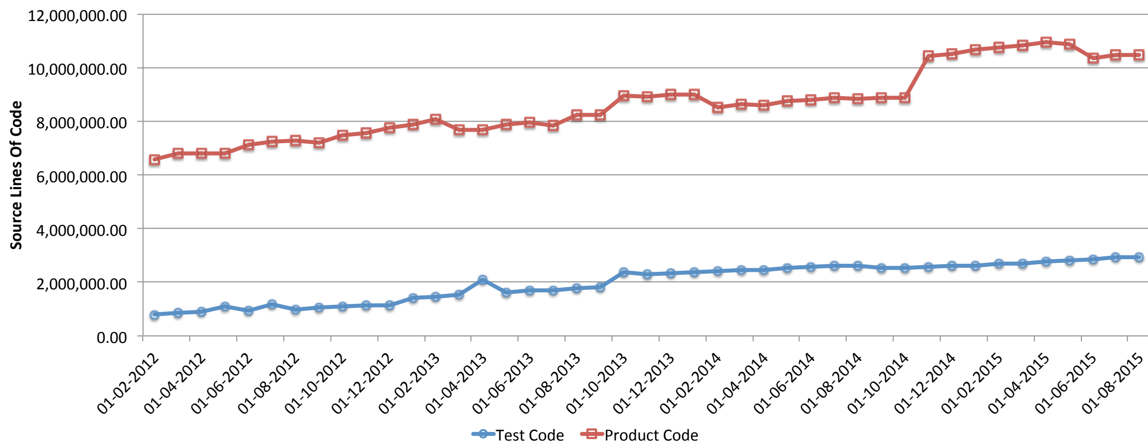


Figure 3.1: Growth of production source code and test source code over the study period. At the beginning, test code accounted for only 10.8% of all code, but by the end this had increased to 21.9%, for a total of over 2.9 million lines of code.

3.3.1 Identifying Build Status

Each time a developer pushes a change or opens a pull request, Travis downloads the change, builds the project, and executes its test suite; this is called a *build*. Travis stores the state of every build, which can have the following values:

- Pass: The build was successfully compiled and all tests in the test suite passed.
- Error: The build failed before test execution began, i.e., there was a compilation or configuration error.
- Fail: The build was successfully compiled, but one or more test assertions failed or encountered an unexpected runtime exception.
- Cancel: A developer manually terminated the build while it was running.
- Started: The build was started but has not finished yet.

Table 3.3 summarizes the number and proportion of builds with each status in our dataset.

Table 3.3: Aggregated build results for 61 open source Java projects that use Travis and the state of 106,738 build executions from those projects.

State	Builds (#)	Builds (%)
Pass	71,303	66.8
Fail	19,640	18.4
Error	15,437	14.5
Cancel	354	0.3
Started	4	0.0
Total	106,738	100.0

3.3.2 Identifying Build Transitions

Though our dataset contained Travis builds, we were primarily interested in how the projects *transitioned* between builds. However, it does not make sense to compare arbitrary commits to each other in a distributed version control system. For instance, if a project has been split into three concurrent branches A, B, and C, it only makes sense to compare adjacent changes in A to each other and adjacent changes in B to each other. Also, if only

one commit is ever made to C, there is no other change to compare it to. Consequently, the number of transitions between builds differs from the number of builds throughout the thesis.

Figure 3.2 shows the transitions between the Travis builds we collected. Due to their relative infrequency we excluded cancelled and started builds as they do not provide any insight into test execution outcomes. Interestingly, the data shows that the build continues to pass only 58% of the time and systems stay in the Pass state for an average of 5.6 builds before transitioning to a Fail or Error state. Error states are fixed more quickly, persisting for an average of 2.6 builds, while Fail states persist for an average of 2.9 builds. These numbers demonstrate that the developers of the Travis projects examined in our study seem to pay attention to test results and fixed issues with their builds in a timely manner. We discuss transitions grouped by their start state in more detail next.

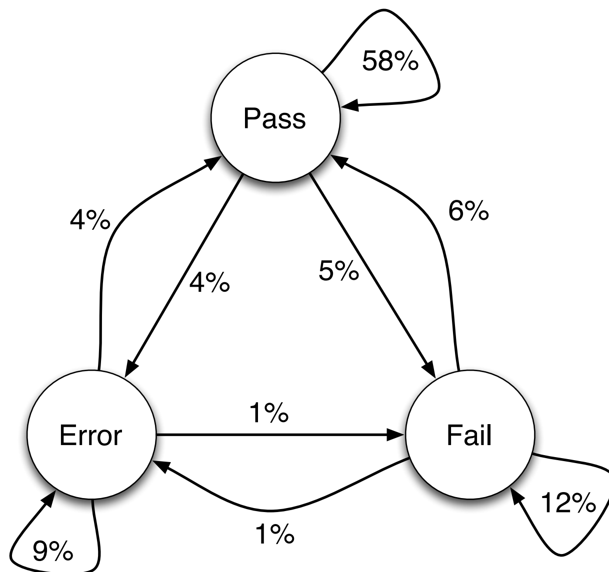


Figure 3.2: Transitions between system states. Each transition is caused by one or more commits. Pass→Pass persisted for an average of 5.6 builds, Error→Error persisted for 2.6 builds, while Fail→Fail builds persisted for 2.9 builds.

Pass → ⋆ The largest proportion of changes in this category correspond to changes where the test suite passes before and after a change. These corresponded to two main cases: changes to a non-code resources (e.g., documentation) or changes to code resources that

did not introduce failures or errors. Builds transitioned into error states when changes were made to either the build infrastructure itself, for instance by changing a configuration file, or when the system could no longer be compiled, which was often caused by missing dependencies. While we expected newly introduced failures to be caused by failure-inducing changes to code resources, we also saw a large proportion of commits that seemed to be failing for no reason at all; further examination of these changes showed that these failure transitions were being caused by flaky tests (see Section 3.4.1).

Fail $\rightarrow \star$ The largest proportion of changes in fail states cause the build to continue to fail; systems in fail states persist for an average of 2.9 builds. We were surprised to see that most systems persisting in failing states seemed to continue to fail because developers were making multiple unrelated changes to the system. While in some cases it seemed that they were aware that these changes were larger and that the build would continue to fail, in other cases they seemed to be disparate changes that happened by chance while the build was already failing. While failing builds sometimes transitioned into error states, primarily through adding new unresolved dependencies, the failures were usually returned to a passing state by fixing the underlying test failures.

Error $\rightarrow \star$ Builds stayed in error states for the shortest period of time (2.6 builds on average). While error states often took a few tries to resolve, mainly by committing changes to configuration files, build scripts, and code dependencies, once these were resolved the build was able to transition back to a passing state. Transitions from the error state to fail state usually corresponded to added dependencies that allowed the code to be compiled so the test suite could run (and fail).

3.3.3 Mapping Builds to Code Changes

In addition to storing build state, Travis stores the branch and SHA of the head commit associated with each build. Note that, as mentioned in Section 3.3.1, Travis builds a project and executes its tests every time a developer pushes a change or opens a GitHub pull request. This means that a Travis build consists of a set of one or more commits; for example, a pull request can be composed of three commits that comprise a single build. We examined the git repository corresponding to each build to determine which commits were part of the Travis build; unfortunately, not all of commits associated with the builds could be reclaimed from the project repository. There are two reasons for this:

1. The build was associated with a pull request. Travis creates a new build every time a pull request is created or updated. The commits associated with these builds, however, are never present in the repository, even if the pull request is merged, as the builds are run against the merge commit between the pull request and the up-stream branch, which does not exist in the master repository.
2. History rewrites. If a developer rewrites history, the build(s) that were triggered by the rewritten commits will no longer be traceable to a commit in the repository.

Ultimately, 70,447 of the 106,738 builds (66%) executed by Travis had associated commits in the project version control repositories. The remainder of our analyses are on these 70,447 builds.

3.3.4 Classifying Failure Resolutions

Having identified builds that transition from the failing state to the passing state, we wanted to know whether this transition was caused by a change to the test code, the system under test, or both. For example, in Figure 3.3, a developer made three consecutive code-only commits to their passing system; in doing so, they introduced a fault that caused a test failure, i.e., a transition to the Fail state. To determine the cause of the failure, we diff the last commit of the Pass build against the last commit of the Fail build. In this case, only source code files were changed, meaning that the cause of the test failure was a source code change (assuming for this example that the failing test(s) were not flaky). The developer then performed three commits that were executed together to return the build to the Pass state; since we cannot determine which commit fixed the fault, we label this a code+test fix.

3.4 Assessing the Costs and Benefits of Regression Testing

Having explained our approach for classifying failure resolutions, we can now answer our first three research questions.

3.4.1 RQ1: What Fraction of Tests Are Flaky?

When the system transitions from the Pass state to the Fail state and back, this indicates that a test failure has occurred and was resolved. There are several possible causes for this.

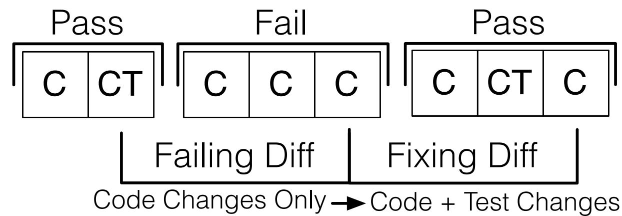


Figure 3.3: As developers work on their systems their commits often change the state of the build. Each box represents a commit; builds are often not run on every commit but instead on blocks of commits. A C label on a commit means the code under test was changed; a T label means the test code was changed. The figure shows a failure caused by a code change that was resolved by fixing both code and test files.

First, the test could be flaky; in this case, the changes made to "resolve" the failure are not actually related to the failure itself. Second, the test could be broken or obsolete; this indicates a cost of regression testing. Third, the test could have identified a failure; this indicates a benefit of regression testing. To determine the cost-effectiveness of the suites under study, it was necessary to identify flaky builds so we could remove them from further analysis.

To determine how common flaky tests are, we examined the 19,640 failing builds in our dataset. We identified build triples in which the system transitioned from the pass state to the failure state and back to the pass state. We excluded all error states as they corresponded to configuration and dependency issues rather than source code defects themselves. For the failing states, we excluded `Fail`$\star$ builds as developers commonly entered these states intentionally when they were landing a diverse set of large changes; in these cases the developers often seemed not to be considering the test suite executions at all as the failures were expected while they worked on changes they knew to be impactful. Ultimately, we found 1,381 instances of builds that underwent `Pass` $\rightarrow$ `Fail` $\rightarrow$ `Pass` transitions. Each instance comprised three builds (by definition) and each build contained an average of 5.2 commits (i.e., 15.6 commits per instance). Figure 3.3 shows a triple matching this pattern consisting of three builds and eight commits.

Having identified these 1,381 triples, we used the Travis API to rerun the failed build for each of the `Pass` $\rightarrow$ `Fail` $\rightarrow$ `Pass` tuples three times. To do this we created three GitHub repositories for each project, each on a separate account, and connected each of these accounts to Travis CI. We then created a branch at the commit associated with each of the failed builds. The branches were then pushed to each of the GitHub repositories,

triggering three identical builds for each branch on Travis CI. To ensure that a build is triggered when a branch is pushed we removed the branch whitelist and blacklist sections from the Travis configuration file and removed the deploy and notification hooks from the configuration file to avoid disturbing the original developers.

If all three re-executions failed we considered the build to be non-flaky. We also considered three passes to be non-flaky, despite a pass being different from the original failure. Only re-executions that included at least two different results were classified as flaky (e.g., failed, failed, passed, or errored, passed, failed). Unfortunately, in many cases all three re-executions errored. We found that this usually happened when dependencies were no longer available or when builds required features no longer supported by Travis CI; these data points amounted to 32% of the re-executions we tried and were discarded. Of the 935 failing builds we successfully re-executed, 120 (12.8%) executed inconsistently and can be considered flaky builds.

Answer 3.1. *13% of the failing builds we studied were flaky.*

3.4.2 RQ2: How Often Are Failures Beneficial?

Having determined which build triples were flaky, we addressed our second research question: which of the non-flaky failures represent a cost of regression testing, and which represent a benefit? Table 3.4 describes all of the possible transitions between three builds that `Pass` → `Fail` → `Pass`. The builds have been grouped by the kind of resource changes that resolved the test failure. From this table, we can see that 58.7% of non-flaky Fail builds are resolved with source code changes alone, 19.3% are resolved by test code changes alone, and 22.0% are fixed by a combination of source and test code changes.

We examined the nine categories of failures in Table 3.4 to determine whether each category represented a cost or benefit of testing. We consider the three categories of failing builds that were resolved by code fixes alone as a positive indication that the test failure identified a source code defect. We make this determination because the developer resolved the failure by changing only the source code; that is, the failing test was correct, and the fix was to modify the system under test to make the test pass in future builds.

In contrast, the three categories of failing builds resolved by test fixes alone represent a cost for the developer: the fault was resolved by maintaining the test suite. These changes acknowledge that the test itself was faulty or obsolete; fixing these failures requires either modifying the test or removing it altogether.

Table 3.4: Categorization of how builds transition between Pass and Fail states. The left-hand-side of → denotes the resources that changed to cause the test failure. The right-hand-side of → denotes the resources that were changed to resolve the test failure. This excludes flaky builds (177 transitions).

Fix Type	Resources Changed	Count	%
Code fixes	Code → Code	527	43.8
	Test → Code	35	2.9
	Code+Test → Code	145	12.0
	∗→ Code	**707**	**58.7**
Test fixes	Code → Test	72	6.0
	Test → Test	47	3.8
	Code+Test → Test	114	9.5
	∗→ Test	**232**	**19.3**
Mixed fixes	Code → Code+Test	100	8.4
	Test → Code+Test	19	1.6
	Code+Test → Code+Test	144	12.0
	∗→ Code+Test	**263**	**22.0**

To gain insight into the nature of the 22.0% of builds that were resolved with both source and test code changes, we examined the proportion of each change that was made to the source code and the test code to fix the build. For these 263 builds, we observed that on average 30% of lines changed were in test files, while the remainder were in source code files. As we considered fixing failing builds by changing the source code beneficial, we classify 70% (15.4% of all changes) of the mixed builds as fixing defects and the remainder (6.6% of all changes) as test maintenance.

Table 3.5 summarizes the proportion of failing builds that are resolved by fixing faults and by maintaining the test suite. These results allow us to answer our second research question.

Answer 3.2. *In the systems under study, 25.9% of non-flaky test suite failures were spurious: they represented developer interruptions that had to be resolved to return the test suite to a Pass state. 74.1% of failures detected real faults.*

It is important to note that we are not claiming that fixing these failing tests could not provide a benefit in the future by detecting faults; however, their maintenance cost must be considered when modelling the overall cost of the testing strategy.

Table 3.5: The proportion of non-flaky Fail builds resolved by fixing faults and by maintaining the test suite.

Change Type	Fix Type	%
Fixing faults	Code only	58.7
	Code+Test	15.4
	Total	**74.1**
Maintaining tests	Test only	19.3
	Code+Test	6.6
	Total	**25.9**

3.4.3 RQ3: Why Do Tests Usually Require Maintenance, and Can Maintenance Costs Be Reduced?

Our previous research question asked what proportion of test failures were resolved by maintaining the failing tests and what proportion of failures were resolved by fixing a real fault. A natural question to ask at this point is, "why do tests frequently require maintenance"? To answer this question, we wanted to identify the tests that required maintenance disproportionately often. Our first step was to identify pass/fail behaviour of individual test cases. We accomplished this by parsing the output associated with the failing build in the non-flaky `Pass` → `Fail` → `Pass` tuples to examine individual test case behaviour. Unfortunately, parsing test logs is a difficult process prone to project-specific noise and one-time errors; due to this we were only able to parse the logs for 40 of the 61 projects.

Once we knew which individual tests passed and failed in each build, we assigned each test case a score: the number of times it failed and caused a code-only fix minus the number of times it failed and caused a test-only fix. A positive number indicates that the test found bugs in the code under test on more occasions than it required maintenance. A negative number is the opposite: the test required maintenance more often than it found bugs.

Figure 3.4 shows the average scores obtained for the tests of 28 projects. The projects `apache/pdfbox` and `apache/jackrabbit-oak` are elided for clarity: the former because it

has a score of 39.25; the latter because its tests experienced 2,416 code and test fixes. Ten projects where the sum of test-only and code-only fixes is smaller than five were removed to reduce clutter. The size of the bubbles indicates the portion of test failures that were resolved by a mixture of code and test changes. This can be seen as the portion of test fixes where there is some doubt as to whether the fix is a return on investment or a maintenance cost. On the x-axis, we plot the sum of all test failures that were resolved by code or test changes alone on a logarithmic scale; this indicates the number of data points we have for each project.

From the figure we can see that, on average, the tests of nine projects require maintenance more often than they find bugs. It is therefore possible that these test suites add very little value or even represent a loss for the projects. We also notice that the difference between code-only fixes and test-only fixes for most projects is small, ranging from -0.87 (`graylog2-server`) to 1.16 (`cloudify`) when excluding the clear outlier, `apache/pdfbox`, where the difference is 39.25. This is in part due to 85% of tests that fail failing only once and 97% failing two times or fewer. There are, however, tests that seem to provide a large return on investment: 24 tests have a score of 5 or higher. There also seem to be tests that may be costly to maintain: 609 tests have a score of -1 and 4 tests a score of -2.

To understand the causes of maintenance, we qualitatively studied the failures associated with seven of the 'least valuable' tests from the data underlying Figure 3.4 to learn why they required an outsized amount of maintenance relative to their defect-detection ability. These tests included the four with a score of -2 and three randomly selected from the 609 tests with a score of -1. The tests are listed in Table 3.6. One of the tests experienced a test-only fix twice, the remaining six experienced only one test-only fix, and none of the tests experienced code-and-test or code-only fixes.

Two of the tests failed due to unintended interaction between tests. The first test (Test 1) depended on an `@after` method deleting the user with email address `JO@FOO.COM` from the database, but failed because the user's email address was in lower case in the database. This test used to work since MySQL, by default, is not case sensitive, but stopped working when the test was run on Postgres which is case sensitive by default. This failure could have been avoided had the developers not depended on the default behaviour of MySQL. The second test (Test 2) was fixed by re-initializing the context between tests. It is, however, interesting that the breaking change removed the re-initialization code and the fix simply replaces it, meaning that it is hard to know why it was removed in the first place.

One of the tests was non-deterministic and from the commit messages we know that the developers were aware of this. The 'fixing' commit for the first test (Test 3) has the commit message "more logging to debug saveTask_shouldSaveTaskToTheDatabase", making it

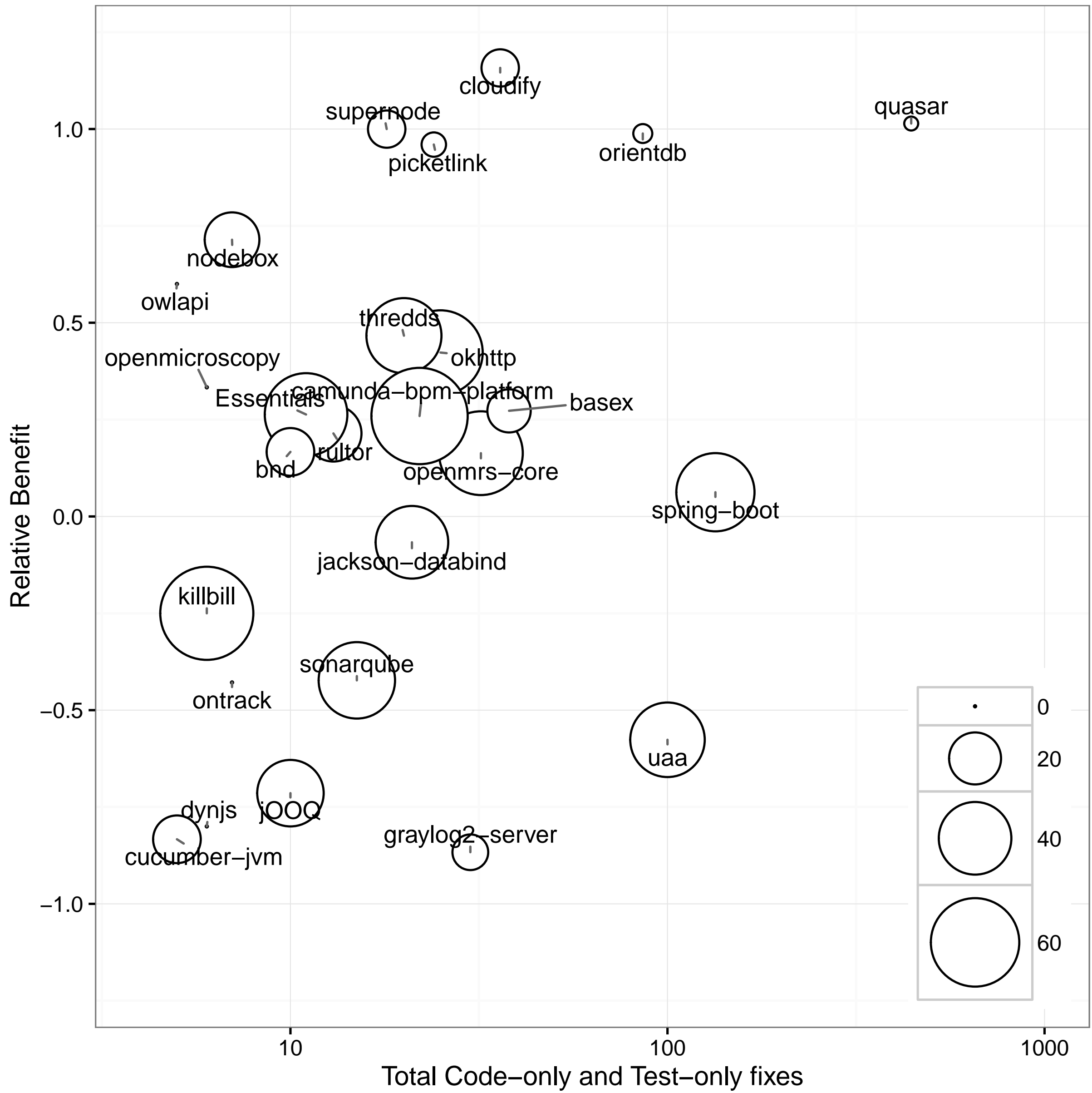


Figure 3.4: Average difference between the number of code fixes and the number of test fixes. The radius of the bubbles represents the relative size of code+test fixes compared to all fixes. The sum of all code fixes and test fixes for a project is plotted on the x-axis on a logarithmic scale.

Table 3.6: Manually inspected tests as well as an example of a broken build and the change that fixed the test. The Break and Fix entries are hyperlinks to the original Travis-CI test suite execution output.

#	Test
(1)	`ProvisioningTests.canCreateUser...` Break: cloudfoundry/uaa/builds/40103431 Fix: cloudfoundry/uaa/builds/40114195
(2)	`PackageAPITest.delete` Break: BaseXdb/basex/builds/1093013 Fix: BaseXdb/basex/builds/1093360
(3)	`SchedulerServiceTest.saveTask...` Break: openmrs/openmrs-core/builds/31049916 Fix: openmrs/openmrs-core/builds/31125642
(4)	`FTIndexQueryTest.testFTTest` Break: BaseXdb/basex/builds/1787079 Fix: BaseXdb/basex/builds/1792995
(5)	`FNClientTest.clientClose` Break: BaseXdb/basex/builds/1434048 Fix: BaseXdb/basex/builds/1434164
(6)	`JavaFuncTest.staticMethod` Break: BaseXdb/basex/builds/848008 Fix: BaseXdb/basex/builds/850290
(7)	`CommandTest` Break: BaseXdb/basex/builds/759351 Fix: BaseXdb/basex/builds/759996

clear that the issue is not yet resolved. When the test fails again at a later stage the ‘fix’ seems to be unrelated to the failing test and we suspect that it does not resolve the issue.

One test (Test 4) was too difficult to analyze and we could not determine whether it is non-deterministic or not. Another test (Test 5) as well as the four others that failed with it are an example of new tests being added to the suite that fail on the first execution. They were fixed by changing a constant.

Of the last two tests, the first is an example of a functional change resulting in test suite maintenance (Test 6) and the second is an accident where changes were applied before they were ready (Test 7).

From these tests, we can see that test failures that lead to maintenance occur for many different reasons. These include tests that fail because of interaction between tests, non-deterministic tests, new tests failing immediately after creation, tests failing after a change in functionality, and an accident where changes were merged before they were ready. This reveals that test code maintenance is not necessarily as tied to product code evolution as one might think. Ensuring that tests do not depend on assumptions (e.g., Test 1) and do not depend on other tests (e.g., Test 2) may reduce the frequency of test maintenance. The prevalence of flaky tests and the developer discussions around them stand out as major costs in automated testing and hinder empirical studies of test results as it is hard to automatically distinguish between deterministic and non-deterministic failures.

Answer 3.3. *"False alarm" test failures have a wide variety of causes, some of which could be avoided, such as dependence on assumptions and on other tests.*

3.5 Implications for Test Selection

In this section, we answer our final research question: what do our data suggest about the limits on test case selection?

3.5.1 RQ4: How Often Do Tests Expose Faults?

The log parsing technique described in the previous section allowed us to count the number of tests that pass and fail on each build to establish the percentage of tests that failed in failing builds. Figure 3.5 shows the proportion of test failures from 40 projects across 586 `Pass` → `Fail` → `Pass` test suite executions from which we could parse the results. From this set, the average test failure rate was 0.38%. It is important to note that this is not the global test case failure rate, but just the test case failure rate within the builds that had at least one test failure.

This number is helpful for understanding the potential effectiveness of test selection approaches as it establishes the absolute minimum fraction of tests a selection approach could execute. That is, if a test selection approach *only* executed tests from builds that would fail, and then *only* executed the failing tests, the approach would have to execute an average of 0.38% of the test suite. Assuming that these failures are spread out equally between the tuple types in Table 3.4 would mean that 25.9% of these test failures did not

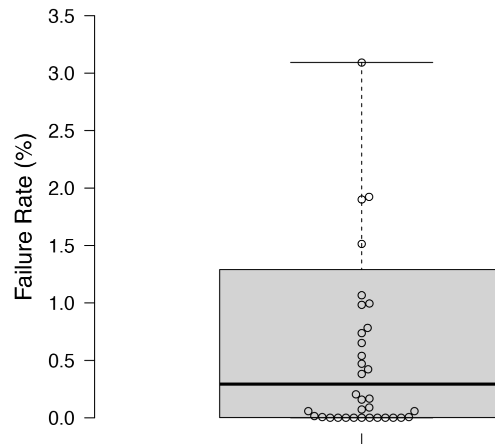


Figure 3.5: Proportion of test cases that fail for the 40 projects we were able to parse individual-test results from across the failing build of 586 tuples. Three data points (16.9%, 8.7% and 4.1%) have been elided for clarity. The average project failure rate was 0.38%.

find faults, bringing the percentage of test case executions that do find faults down to 0.28%.

Interestingly, we found that 64% of failed builds contain more than one failed test. These failures usually have the same root cause and therefore all the failed tests were not necessary to find the fault. Test selection strategies typically use coverage overlap to determine whether a test is redundant or not, but this alone can be inaccurate [60]. By only considering a test case redundant if all past failures occurred with other tests the false-positive rate could be reduced. Conversely, if a test has failed alone it would indicate that it is not redundant even if it does have complete coverage overlap with another test.

It is also worth considering that multiple failing tests could provide additional insight into the underlying cause of a failure. For example, suppose a number of tests related to sending an email fail. These tests might differ in whether they include an attachment, use special formatting, and so on. When they all fail together, the developer may realize that the mail server is down and thus the root cause of the problem is different from that suggested by any one test.

Answer 3.4. *In the failed builds under study, only 0.28% of test case executions failed due to a fault. Only 36% of build failures are caused by a single test case failing.*

3.6 Threats to Validity

The dataset presented in Section 3.2 has a number of limitations, most notably to external validity. While it presents a diverse set of 61 projects, they were mostly Java-based projects that used the Travis continuous integration platform. In the process of trying to find representative projects, we also filtered out projects that we believed to be unusual, but whose test-inducing behaviour may have been interesting. That said, the set of projects was diverse and contained test suites that were being actively maintained and executed. The projects may have used different development processes, which would increase project diversity but make comparisons between projects less meaningful.

One may also wonder whether developers run some tests locally before committing to the repository, skewing our results. This is a common threat in studies that mine software repositories: one may be exploring development as it was recorded, rather than as it happened. We do not believe developers ran tests locally for three reasons. First, we eliminated the projects with the fewest build failures; i.e., those that mirror stable repositories. Second, of the remaining projects, the large number of error and failure builds suggest that developers frequently break the build; that is, they do not worry about ensuring the code can be compiled or tested before committing. Third, it is unlikely that developers would set up CI infrastructure and then run tests manually. We feel that, just as developers who set up Ant or Maven are unlikely to run `javac` on the command line, developers who set up Travis are unlikely to replicate its functionality locally.

Our goal in Section 3.4 was to measure how often a test suite failure indicated that test suite maintenance was necessary. Though it seems reasonable to assume that, if only the test code was changed to fix a test failure, the developers were maintaining the test, it is possible that we misclassified some of the test failures. We note that we did not see any such misclassifications during our manual inspection of tests. However, examining each change manually would have been infeasible, and feel our approach is an adequate approximation.

Additionally, in the code fixes category, some of the Code+Test changes could have involved test maintenance that was done before the tests failed because the developer knew that their code changes would cause test failures. Thus, in terms of internal validity, these analyses may be overly conservative as we have excluded some test changes that could have been classified as maintenance.

Our results for the flaky analysis were also limited: while we found that 12.8% of the tuples we examined were flaky, we examined only builds that failed when the developer originally executed them. It is possible for flaky tests to pass by chance, and thus some of the passing builds we did not re-execute could have been flaky as well.

Finally, though we compared the *number* of times tests were maintained and the *number* of times they detected faults, we did not attempt to do any cost estimation. A test that frequently requires maintenance but detects one critical fault may be more valuable than one that rarely requires maintenance but only detects minor faults. However, on average, more maintenance means a more expensive test suite while more faults detected means the suite is providing more benefit.

3.7 Replication Package

The data underlying this paper represents an oracle consisting of code changes and test failures that arose in practice. This dataset is valuable because it augments past studies on industrial code for which the full data could not be released (e.g., [23, 41, 46]). It also provides crucial information about the dynamic outcomes of test executions that cannot be recovered from mining studies alone (e.g., [13, 70, 90]).

The dataset includes a database image of the build results of 225,860 Travis builds run by 493 projects (some projects not studied in this paper are included). Build results include the build state, timestamps, and, for builds whose logs we could parse, the test identifiers of failed tests. Unparsed logs of all builds are also provided in text files.

These data represent the largest open collection of practical test failures and will prove valuable for many future studies that want to evaluate their approaches on real data by, for instance, measuring how many actual failures would be missed by a test minimization approach, or how effective a test prioritization approach is relative to a known 'best case'. The full replication package for this study can be found and contributed to online.[5]

3.8 Relevance to Thesis

Recall that our thesis statement is as follows:

Thesis Statement. *Robust development processes are necessary to minimize the number of faults introduced when evolving complex software systems. These processes should be based on empirical research findings. Data science techniques allow software engineering researchers to develop research insights that may be difficult or impossible to obtain with other research methodologies. These research insights support the creation of development processes. Thus, data science techniques support the creation of empirically-based development processes.*

[5] `github.com/rtholmes/RealTestFailures/`

Writing and executing automated tests is a development process that can improve code quality. However, maintaining such a test suite is costly. To determine whether and to what extent to adopt automated testing, developers need to carefully consider both the costs and the benefits of their test suite. Our work provides insight into this tradeoff by showing that, for the systems we studied, 26% of non-flaky test failures represented a maintenance cost and an individual test execution had only a 0.28% chance of failing and exposing a real defect for any given code change. It is difficult to use this information to make general recommendations for developers, as the testing budget and the ramifications of faults that are not detected depend on the project. However, having these numbers in hand may allow a developer to convince other developers or management to measure the costs and benefits of their own test suite. In addition, our findings indicate that current test case selection approaches can still be improved.

Information about the cost/benefit tradeoff of test suites could be obtained using other research methodologies. For example, one could visit a software company, instrument the IDEs being used by the developers in order to record the amount of time spent running tests, and ask the developers to self-report the number of faults detected. Though this would work, we would not have been able to get data about 61 projects and therefore our study would lack external generalizability. The use of data science techniques allowed us to get a research result that would have been infeasible to obtain via other means. Thus, the work in this chapter provides an example of the ways in which the use of data science techniques can allow researchers to develop new insights that in turn support software developers.

Chapter 4

The Use of Mutants in Testing Research

Both industrial software developers and software engineering researchers are interested in measuring test suite effectiveness. While developers want to know whether their test suites have a good chance of detecting faults, researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown *a priori*, so a proxy measurement must be used instead.

A well-established proxy measurement for test suite effectiveness in testing research is the *mutation score*, which measures a test suite's ability to distinguish a program under test, the *original version*, from many small syntactic variations, called *mutants*. Specifically, the mutation score is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, using well-defined *mutation operators*. Examples of such mutation operators are replacing arithmetic or relational operators, modifying branch conditions, or deleting statements (cf. [48]).

Mutation analysis is often used in software testing and debugging research. More concretely, it is commonly used in the following use cases (e.g., [11,25,48,49,97,105,107,110]):

Test suite evaluation The most common use of mutation analysis is to evaluate and compare (generated) test suites. Generally, a test suite that has a higher mutation score is assumed to detect more real faults than a test suite that has a lower mutation score.

Test suite selection Suppose two unrelated test suites T_1 and T_2 exist that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score.

Test suite minimization A mutation-based test suite minimization approach reduces a test suite T to $T \setminus \{t\}$ for every test $t \in T$ for which removing t does not decrease the mutation score of T.

Test suite generation A mutation-based test generation (or augmentation) approach aims at generating a test suite with a high mutation score. In this context, a test generation approach augments a test suite T with a test t only if t increases the mutation score of T.

Fault localization A fault localization technique that precisely identifies the root cause of an artificial fault, i.e., the mutated code location, is assumed to also be effective for real faults.

These uses of mutation analysis rely on the assumption that mutants are a valid substitute for real faults. Unfortunately, there is little experimental evidence supporting this assumption, as discussed in greater detail in Section 4.3. To the best of our knowledge, only three previous studies have explored the relationship between mutants and real faults [6, 19, 79]. Our work differs from these previous studies in four main aspects. (1) Our study considers subject programs that are orders of magnitude larger. (2) Our study considers real faults rather than hand-seeded faults. (3) Our study uses developer-written and automatically-generated test suites. (4) Our study considers the conflating effects of code coverage when studying the correlation between mutant detection and real fault detection. A higher mutant detection and real fault detection rate could both be caused by higher code coverage, thus it is important to control this variable when measuring the correlation.

Specifically, this paper extends previous work and explores the relationship between mutants and real faults using 5 large Java programs, 357 real faults, and 230,000 mutants. It aims to confirm or refute the hypothesis that mutants are a valid substitute for real faults in software testing by answering the following questions:

Research Question 4.1. *Are real faults coupled to mutants generated by commonly used mutation operators?*

The existence of the *coupling effect* [20] is a fundamental assumption underlying mutation analysis. A complex fault is *coupled* to a set of simple faults if a test that detects all the simple faults also detects the complex fault. Prior research empirically showed the

existence of the coupling effect between simple and complex mutants [47, 82], but it is unclear whether real faults are coupled to simple mutants derived from commonly used mutation operators [48, 80, 83]. Therefore, this paper investigates whether this coupling effect exists. In addition, it studies the numbers of mutants coupled to each of the real faults as well as their underlying mutation operators.

Research Question 4.2. *What types of real faults are not coupled to mutants?*

The coupling effect may not hold for every real fault. Therefore, this paper investigates what types of real faults are not coupled to any of the generated mutants. Additionally, this paper sheds light on whether the absence of the coupling effect indicates a weakness of the set of commonly applied mutation operators or an inherent limitation of mutation analysis.

Research Question 4.3. *Is mutant detection correlated with real fault detection?*

Since mutation analysis is commonly used to evaluate and compare (generated) test suites, this paper also addresses the question of whether a test suite's ability to detect mutants is correlated with its ability to detect real faults.

In summary, the contributions of this paper are as follows:

- A new set of 357 developer-fixed and manually-verified real faults and corresponding test suites from 5 programs.
- The largest study to date of whether mutants are a valid substitute for real faults using 357 real faults, 230,000 mutants, and developer-written and automatically-generated tests.
- An investigation of the coupling effect between real faults and the mutants that are generated by commonly used mutation operators. The results show the existence of a coupling effect for 73% of real faults.
- Concrete suggestions for improving mutation analysis (10% of real faults require a new or stronger mutation operator), and identification of its inherent limitations (17% of real faults are not coupled to mutants).
- An analysis of whether mutant detection is correlated with real fault detection. The results show a statistically significant correlation that is stronger than the correlation between statement coverage and real fault detection.

The remainder of this chapter is structured as follows. Section 4.1 describes our method and the experiments we performed to answer our research questions. Section 4.2 presents and discusses the results. Section 4.3 reviews related work, and Section 4.4 explains how this study supports our thesis statement.

Table 4.1: Investigated subject programs.

Program size (KLOC), test suite size (Test KLOC), and the number of JUnit tests (Tests) are reported for the most recent version. LOC refers to non-comment, non-blank lines of code and was measured with sloccount (`http://www.dwheeler.com/sloccount`).

	Program	**KLOC**	**Test KLOC**	**Tests**
Chart	JFreeChart	96	50	2,205
Closure	Closure Compiler	90	83	7,927
Math	Commons Math	85	19	3,602
Time	Joda-Time	28	53	4,130
Lang	Commons Lang	22	6	2,245
Total		321	211	20,109

4.1 Method

Our goal was to test the assumption that mutants are a valid substitute for real faults by conducting a study with real faults, using both developer-written and automatically-generated test suites. To accomplish this, we performed the following high-level steps:

1. Located and isolated real faults that have been previously found and fixed by analyzing the subject programs' version control and bug systems (Section 4.1.2).
2. Obtained developer-written test suites for both the faulty and the fixed program version for each real fault (Section 4.1.3).
3. Automatically generated test suites for the fixed program version for each real fault (Section 4.1.4).
4. Generated mutants and performed mutation analysis for all fixed program versions (Section 4.1.5).
5. Conducted experiments using the real faults, mutants, and the test suites to answer our research questions (Section 4.1.6).

4.1.1 Subject Programs

Table 4.1 lists the 5 subject programs we used in our experiments. These programs satisfy the following desiderata:

1. Each program has a version control and bug tracking system, enabling us to locate and isolate real faults.

2. Each program is released with a comprehensive test suite, enabling us to experiment with developer-written test suites in addition to automatically-generated ones.

4.1.2 Locating and Isolating Real Faults

We obtained real faults from each subject program's version control system by identifying commits that corrected a failure in the program's source code. Ideally, we would have obtained, for each real fault, two source code versions V_{bug} and V_{fix} which differ by only the bug fix. Unfortunately, developers do not always minimize their commits. Therefore, we had to locate and isolate the fix for the real fault in a bug-fixing commit.

We first examined the version control and bug tracking system of each program for indications of a bug fix (Section 4.1.2). We refer to a revision that indicates a bug fix as a *candidate revision*. For each candidate revision, we tried to reproduce the fault with an existing test (Section 4.1.2). Finally, we reviewed each reproducible fault to ensure that it is isolated, i.e., the bug-fixing commit does not include irrelevant code changes (Section 4.1.2). We discarded any fault that could not be reproduced and isolated. Table 4.2 summarizes the results of each step in which we discarded candidate revision pairs.

Candidate Revisions for Bug-Fixing Commits

We developed a script to determine revisions that a developer marked as a bug fix. This script mines the version control system for explicit mentions of a bug fix, such as a bug identifier from the subject program's bug tracking system.

Let rev_{fix} be a revision marked as a bug fix. We assume that the previous revision in the version control system, rev_{bug}, was faulty (later steps validate this assumption). Overall, we identified 1,179 candidate revision pairs $\langle rev_{bug}, rev_{fix} \rangle$.

Discarding Non-reproducible Faults

A candidate revision pair obtained in the previous step is not suitable for our experiments if we cannot reproduce the real fault. Let V be the source code version of a revision *rev*, and let T be the corresponding test suite. The fault of a candidate revision pair $\langle rev_{bug}, rev_{fix} \rangle$ is reproducible if a test exists in T_{fix} that passes on V_{fix} but fails on V_{bug} due to the existence of the fault.

In some cases, test suite T_{fix} does not run on V_{bug}. If necessary, we fixed build-system-related configuration issues and trivial errors such as imports of non-existent classes.

Table 4.2: Number of candidate revisions, compilable revisions, and reproducible and isolated faults for each subject program.

	Candidate revisions	Compilable revisions	Reproducible faults	Isolated faults
Chart	80	62	28	26
Closure	316	227	179	133
Math	435	304	132	106
Time	75	57	29	27
Lang	273	186	69	65
Total	1179	836	437	357

However, we did not attempt to fix compilation errors requiring non-trivial changes, which would necessitate deeper knowledge about the program. 836 out of 1,179 revision pairs remained after discarding candidate revision pairs with unresolvable compilation errors.

After fixing trivial compilation errors, we discarded version pairs for which the fault was not reproducible. A fault might not be reproducible for three reasons. (1) The source code diff is empty — the difference between rev_{bug} and rev_{fix} was only to tests, configuration, or documentation. (2) No test in T_{fix} passes on V_{fix} but fails on V_{bug}. (3) None of the tests in T_{fix} that fail on V_{bug} exposes the real fault. We manually inspected each test of T_{fix} that failed on V_{bug} while passing on V_{fix} to determine whether its failure was caused by the real fault. Examples of failing tests that do not expose a real fault include dependent tests [119] or non-deterministic tests. The overall number of reproducible candidate revision pairs was 437.

Discarding Non-isolated Faults

Since developers do not always minimize their commits, the source code of V_{bug} and V_{fix} might differ by both features and the bug fix. We ensured that all bug fixes were isolated for the purposes of our study. Isolation is important because unrelated changes could affect the outcome of generated tests or could affect the coverage and mutation score. Other benefits of isolation include improved backward-compatibility of tests and the ability to focus our experiments on a smaller amount of modified code.

For each of the 437 reproducible candidate revision pairs, we manually reviewed the bug fix (the source code diff between V_{bug} and V_{fix}) to verify that it was isolated and related to

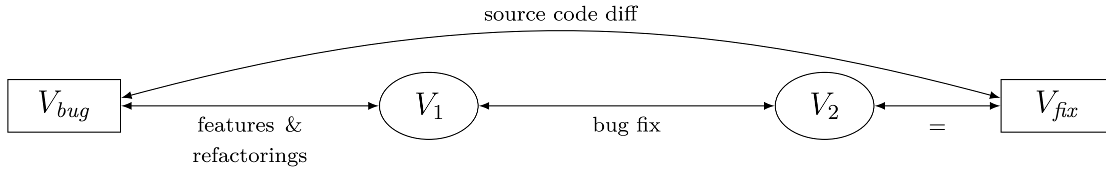


Figure 4.1: Obtaining source code versions V_1 and V_2.
V_1 and V_2 differ by only a bug fix. V_{bug} and V_{fix} are the source code versions of two consecutive revisions in a subject program's version control system.

the real fault. We divided a non-isolated bug fix into two diffs, one that represents the bug fix and one that represents features and refactorings. We discarded a candidate revision pair if we could not isolate the bug fix part of the diff. The result of this step was two source code versions V_1 and V_2 such that V_1 and V_2 differ by exactly a bug fix — no features were added and no refactorings were applied. To ensure consistency, the review process was performed twice by different authors, with a third author resolving disagreements. Different authors reviewed different diffs to avoid introducing a systematic bias.

Figure 4.1 visualizes the relationship between the source code versions V_1 and V_2, and how they are obtained from the source code versions of a candidate revision pair. V_2 is equal to the version V_{fix}, and the difference between V_1 and V_2 is the bug fix. Note that V_1 is obtained by re-introducing the real fault into V_2 — that is, applying the inverse bug-fixing diff. Overall, we obtained 357 version pairs $\langle V_1, V_2 \rangle$ for which we could isolate the bug fix.

4.1.3 Obtaining Developer-written Test Suites

Section 4.1.2 described how we obtained 357 suitable version pairs $\langle V_1, V_2 \rangle$. This section describes how we obtained two related test suites T_{pass} and T_{fail} made up of developer-written tests. T_{pass} and T_{fail} differ by exactly one test, T_{pass} passes on V_1, and T_{fail} fails on V_1 because of the real fault.

Since T_{pass} and T_{fail} differ by exactly one test related to the real fault, the pairs $\langle T_{pass}, T_{fail} \rangle$ enable us to study the coupling effect between real faults and mutants, and whether the effect exists independently of code coverage. These test suite pairs also reflect common and recommended practice. The developer's starting point is the faulty source code version V_1 and a corresponding test suite T_{pass}, which passes on V_1. Upon discovering a previously-unknown fault in V_1, a developer augments test suite T_{pass} to expose this fault.

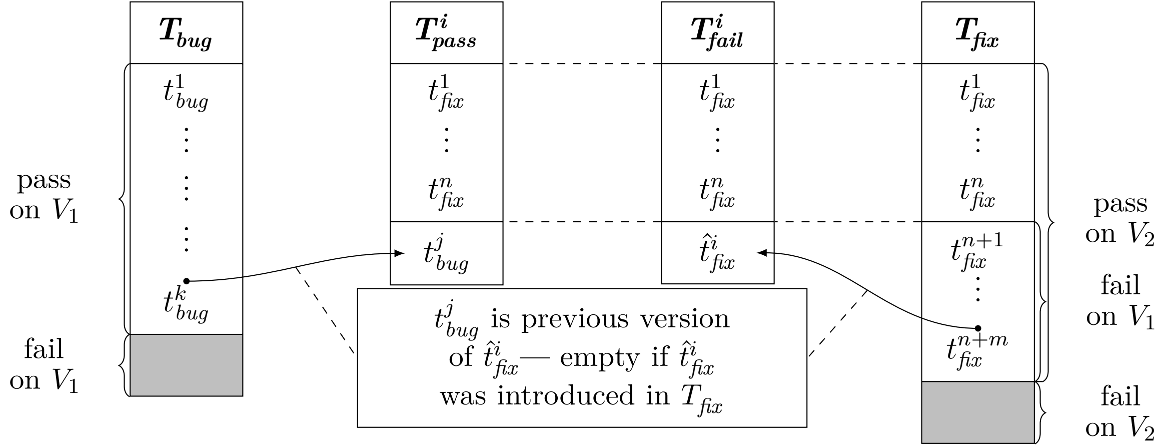


Figure 4.2: Relationship between the i-th obtained test suite pair $\langle T^i_{pass}, T^i_{fail} \rangle$ and the developer-written test suites T_{bug} and T_{fix}.

T_{bug} and T_{fix} are derived from a subject program's version control system. $\hat{t}^i_{fix}$ is the i-th triggering test in T_{fix}, and t^j_{bug} is the previous version of that test.

The resulting test suite T_{fail} fails on V_1 but passes on the fixed source code version V_2. T_{pass} might be augmented by modifying an existing test (e.g., adding stronger assertions) or by adding a new test.

We cannot directly use the existing developer-written test suites T_{bug} and T_{fix} as T_{pass} and T_{fail}, because not all tests pass on each committed version and because the developer may have committed changes to the tests that are irrelevant to the fault. Therefore, we created the test suites T_{pass} and T_{fail} based on T_{bug} and T_{fix}, as we now describe.

Recall that for each pair $\langle V_1, V_2 \rangle$, one or more tests expose the real fault in V_1 while passing on V_2 — we refer to such a test as a *triggering test*, $\hat{t}$. Let m be the number of triggering tests for a version pair; then $\hat{t}^i$ denotes the i-th triggering test ($1 \leq i \leq m$). Figure 4.2 visualizes how we obtained, for each real fault, m pairs of test suites $\langle T^i_{pass}, T^i_{fail} \rangle$ with the following properties:

- T^i_{pass} passes on V_1 and V_2.
- T^i_{fail} fails on V_1 but passes on V_2.
- T^i_{pass} and T^i_{fail} differ by exactly one modified or added test.

In order to fairly compare the effectiveness of T_{pass} and T_{fail}, they must not contain irrelevant differences. Therefore, T_{pass} is derived from T_{fix}. If T_{pass} were derived from T_{bug} instead, two possible problems could arise. First, V_1 might include features (compared

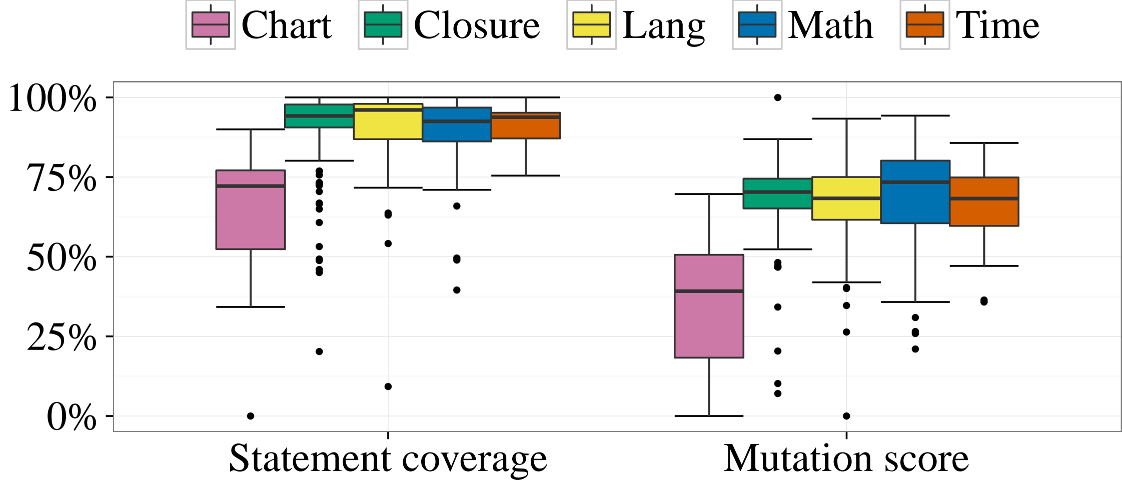


Figure 4.3: Statement coverage ratios and mutation scores of the test suites T_{pass} for each subject program.

to V_{bug}, as described in Section 4.1.2), and T_{fix} might include corresponding feature tests. Second, tests unrelated to the real fault might have been added, changed, or removed in T_{fix}.

In summary, we applied the following steps to obtain all pairs $\langle T_{pass}, T_{fail} \rangle$ using the developer-written test suites T_{bug} and T_{fix}:

1. Manually fixed all classpath- and configuration-related test failures in T_{bug} and T_{fix} to ensure that all failures indicate genuine faults.
2. Excluded all tests from T_{bug} that fail on V_1, and excluded all tests from T_{fix} that fail on V_2.
3. Determined all triggering tests $\hat{t}^i_{fix}$ in T_{fix}.
4. Created one test suite pair $\langle T^i_{pass}, T^i_{fail} \rangle$ for each $\hat{t}^i_{fix} \in T_{fix}$ (as visualized in Figure 4.2).

Overall, we obtained 480 test suite pairs $\langle T_{pass}, T_{fail} \rangle$ in this step. Figure 4.3 summarizes the statement coverage ratios and mutation scores for all test suites T_{pass} measured for classes modified by the bug fix. The high degree of statement coverage achieved by T_{pass} allowed us to obtain 258 test suite pairs for which coverage did not increase and 222 test suite pairs for which it did.

In 80% of the cases, T_{fix} contained exactly one triggering test; developers usually augment a test suite by adding or strengthening one test to expose the fault. For the remaining cases, each triggering test exposes the real fault differently. For example, a developer might add two tests for a boundary condition bug fix – one test to check the maximum and one test to check the minimum value.

4.1.4 Automatically Generating Test Suites

We used three test generation tools for our study: EvoSuite [24], Randoop [86], and JCrasher [17]. We attempted to use DSDCrasher [18] instead of JCrasher, but found that it relies on the static analysis tool ESC/Java2. This tool does not work with Java 1.5 and higher, making it impossible to use DSDCrasher for this study.

Unlike Randoop and JCrasher, EvoSuite aims to satisfy one of three possible criteria — branch coverage, weak mutation testing, or strong mutation testing. We generated tests for each of the criteria. We also selected two different configurations for Randoop, one that allows `null` values as inputs (Randoop-null) and one that does not (Randoop-nonnull). For each fixed program version V_2, we generated 30 test suites with EvoSuite for each of the selected criteria, 6 test suites for each configuration of Randoop, and 10 test suites with JCrasher. Each test generation tool was guided to create tests only for classes modified by the bug fix.

Each of the test generation tools might produce tests that do not compile or do not run without errors. Additionally, tests might sporadically fail due to the use of non-deterministic APIs such as time of day or random number generators. A test suite that (sporadically) fails is not suitable for our study. We automatically repaired uncompilable and failing test suites using the following workflow:

1. Removed all tests that cause compilation errors.
2. Removed all tests that fail during execution on V_2.
3. Iteratively removed all non-deterministic tests; we assumed that a test suite does not include any further non-deterministic tests once it passed 5 times in a row.

The final output of this process was generated test suites that pass on V_2. Repairing a test suite resulted in approximately 2% of cases in an empty test suite, when all tests failed and had to be removed. Therefore, for all tools used to generate tests, the number of suitable test suites, which pass on V_2, is smaller than the total number of generated test suites. Table 4.3 summarizes the characteristics of all generated test suites that pass on V_2. Note that unlike EvoSuite and Randoop, JCrasher does not capture program behavior for

Table 4.3: Characteristics of generated test suites.

Test suites gives the total number of test suites that passed on V_2 and the percentage of test suites that detected a real fault (T_{fail}). The KLOC and Tests columns report the mean and standard deviation of lines of code and number of JUnit tests for all test suites. Detected faults shows how many distinct real faults the test suites detected out of the number of program versions for which at least one suitable test suite could be generated.

	Test suites		**KLOC**	**Tests**	**Detected faults**
	Total	T_{fail}			
EvoSuite	28,318	22.3%	10±49	68±133	182/354
-branch	10,133	21.1%	2±7	21±24	156/352
-weak	9,420	21.8%	3±8	24±27	158/352
-strong	8,765	24.1%	26±86	171±202	152/350
Randoop	3,387	18.0%	212±132	6,929±9,923	90/326
-nonnull	1,690	17.3%	200±124	6,113±9,012	78/316
-null	1,697	18.7%	224±138	7,747±10,698	84/319
JCrasher	3,436	0.6%	543±561	47,928±48,174	2/350
Total	35,141	19.7%	335±995	1,066±5,599	198/357

regression testing but rather aims at crashing a program with an unexpected exception, explaining the low real fault detection rate.

We executed each generated test suite $\tilde{T}$ on V_1. If it passed ($\tilde{T}_{pass}$), it did not detect the real fault. If it failed ($\tilde{T}_{fail}$), we verified that the failing tests are valid triggering tests, i.e., they do not fail due to build system or configuration issues. Overall, the test generation tools created 35,141 test suites that detect 198 of the 357 real faults. Figure 4.4 gives the statement coverage ratios and mutation scores for all generated test suites grouped by subject program, test generation tool/configuration, and real fault detection rate.

4.1.5 Mutation Analysis

We used the Major mutation framework [50, 51] to create the mutant versions and to perform the mutation analysis. Major provides the following set of mutation operators: *Replace constants*, *Replace operators*, *Modify branch conditions*, and *Delete statements*. This set, suggested in the literature on mutation analysis [48, 55, 80, 83], is commonly used and includes the mutation operators used by previous studies [6, 19].

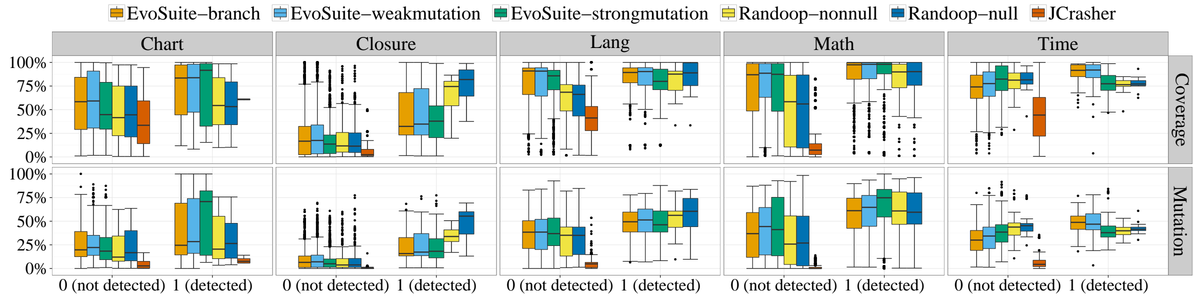


Figure 4.4: Statement coverage ratios and mutation scores of the generated test suites for each subject program.
The vertical axis shows the statement coverage ratio (Coverage) and the mutation score (Mutation). The horizontal axis shows the real fault detection rate.

Major only mutated classes of the source code version V_2 that were modified by the bug fix. This reduces the number of mutants irrelevant to the fault – differences in the mutation score would be washed out otherwise.

For each of the developer-written and automatically-generated test suites, Major computed mutation coverage and mutation score. A test is said to cover a mutant if it reaches and executes the mutated code. A test detects a mutant if the test outcome indicates a fault — a test assertion fails or the test causes an exception in the mutant.

We did not eliminate equivalent mutants, which means that the reported mutation scores might be underestimated. This is, however, not a concern for our study because we do not interpret absolute mutation scores. Moreover, the set of equivalent mutants is identical for any two test suites used in a comparison.

4.1.6 Experiments

As described in the introduction, the goal of our study was to answer three research questions:

1. Are real faults coupled to mutants generated by commonly used mutation operators?
2. What types of real faults are not coupled to mutants?
3. Is mutant detection correlated with real fault detection?

After explaining why and how we controlled for code coverage, this section explains how we answered these three questions.

Controlling for Code Coverage

Structural code coverage is a widely-used measure of test suite effectiveness. Differences in coverage often dominate other aspects of test suite effectiveness, and a test suite that achieves higher coverage usually detects more mutants and faults for that reason alone [43]. More specifically, if test suite T_x covers more code than T_y, then T_x is likely to have a higher overall mutation score, even if T_y does a better job in testing a smaller portion of the program.

Furthermore, no developer would use a complex, time-consuming test suite metric such as the mutation score unless simpler ones such as structural code coverage ratios had exhausted their usefulness.

To account for these facts, we performed our experiments in two ways. First, we ignored code coverage and simply determined the mutation score for each test suite using all mutants. Second, we controlled for coverage and determined the mutation score using only mutants in code covered by both test suites.

For the related test suite pairs $\langle T_{pass}, T_{fail} \rangle$, T_{pass} and T_{fail} may have the same code coverage: T_{pass} and T_{fail} cover the same code if the triggering test in T_{fail} does not increase code coverage.

For the automatically-generated test suites, it is highly unlikely that $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$ have the same coverage because they were independently generated. Therefore, we had to control for coverage when using the automatically-generated test suites. We did this by only considering the intersection of mutants covered by both test suites. This means that a pair of generated test suites was discarded if the intersection was the empty set.

We include the first, questionable method for comparison with prior research that does not control for coverage. The second method controls for coverage. It better answers whether use of mutation analysis is profitable, under the assumption that a developer is already using the industry-standard coverage metric. Our experiments use Cobertura [16] to compute statement coverage over the classes modified by the bug fix.

Are Real Faults Coupled to Mutants Generated by Commonly Used Mutation Operators?

The test suites T_{pass} and T_{fail} model how a developer usually augments a test suite. T_{fail} is a better suite — it detects a fault that T_{pass} does not. If mutants are a valid substitute for real faults, then any test suite T_{fail} that has a higher real fault detection rate than

T_{pass} should have a higher mutation score as well. In other words, each real fault should be coupled to at least one mutant. For each test suite pair $\langle T_{pass}, T_{fail} \rangle$, we studied the following questions:

- Does T_{fail} have a higher mutation score than T_{pass}?
- Does T_{fail} have a higher statement coverage than T_{pass}?
- Is the difference between T_{pass} and T_{fail} a new test?

Based on the observations, we performed three analyses. (1) We used the Chi-square test to determine whether there is a significant association between the measured variables *mutation score increased*, *statement coverage increased*, and *test added*. (2) We determined the number of real faults coupled to at least one of the generated mutants. (3) We measured the sensitivity of the mutation score with respect to the detection of a single real fault – the increase in the number of detected mutants between T_{pass} and T_{fail}. We also determined the mutation operators that generated the mutants additionally detected by T_{fail}. Section 4.2.1 discusses the results.

What Types of Real Faults Are Not Coupled to Mutants?

Some of the real faults are not coupled to any of the generated mutants, i.e., the set of mutants detected by T_{pass} is equal to the set of mutants detected by T_{fail}. We manually investigated each such fault. This qualitative study reveals how the set of commonly used mutation operators should be improved. Moreover, this study shows what types of real faults are not coupled to any mutants and therefore reveals general limitations of mutation analysis. Section 4.2.2 discusses the results.

Is Mutant Detection Correlated with Real Fault Detection?

We conducted two experiments to investigate whether a test suite's mutation score is correlated with its real fault detection rate. Calculating the correlation requires larger numbers of test suites per fault, and thus we used the automatically-generated test suites. We analyzed the entire pool of test suites derived from all test generation tools to investigate whether the mutation score is generally a good metric to compare the effectiveness of arbitrary test suites. The experiments consider 194 real faults for which we could generate at least one test suite that detects the real fault and at least one test suite that does not.

We determined the strength of the correlation between mutation score and real fault detection. Since real fault detection is a dichotomous variable, we computed the point-biserial and rank-biserial correlation coefficients. In addition, we investigated whether the

correlation is significantly stronger than the correlation between statement coverage and real fault detection.

While we cannot directly calculate the correlation between mutation score and real fault detection independently of code coverage, we can still determine whether there is a statistically significant difference in the mutation score between $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$ when coverage is fixed. Calculating the correlation coefficient independently of code coverage would require fixed coverage over all test suites. In contrast, testing whether the mutation score differs significantly requires only fixed coverage between pairs of test suites.

For each real fault, we compared the mutation scores of $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$. Since the differences in mutation score were not normally distributed (evaluated by the Kolmogorov-Smirnov test), a non-parametric statistical test was required. Using the Wilcoxon signed-rank test, we tested whether the mutation scores of $\tilde{T}_{fail}$ are significantly higher than the mutation scores of $\tilde{T}_{pass}$, independently of code coverage. Additionally, we measured the $\hat{A}_{12}$ effect sizes for the mutation score differences. Section 4.2.3 discusses the results.

4.2 Results

Section 4.1 described our method and analyses. This section answers the posed research questions. Recall that we used 357 real faults, 480 test suite pairs $\langle T_{pass}, T_{fail} \rangle$ made up of developer-written tests, and 35,141 automatically-generated test suites which may ($\tilde{T}_{fail}$) or may not ($\tilde{T}_{pass}$) detect a real fault.

4.2.1 Are Real Faults Coupled to Mutants Generated by Commonly Used Mutation Operators?

Considering all test suite pairs $\langle T_{pass}, T_{fail} \rangle$, the mutation score of T_{fail} increased compared to T_{pass} for 362 out of 480 pairs (75%). Statement coverage increased for only 222 out of 480 pairs (46%).

The mutation score of T_{fail} increased for 153 out of 258 pairs (59%) for which statement coverage did not increase. The mutation score of T_{fail} increased for 209 out of 222 pairs (94%) for which statement coverage increased. The Chi-square test showed a significant association between *mutation score increased* and *statement coverage increased* ($\chi^2(1) = 78.13, N = 480, p < 0.001$), hence we considered the influence of statement coverage throughout our

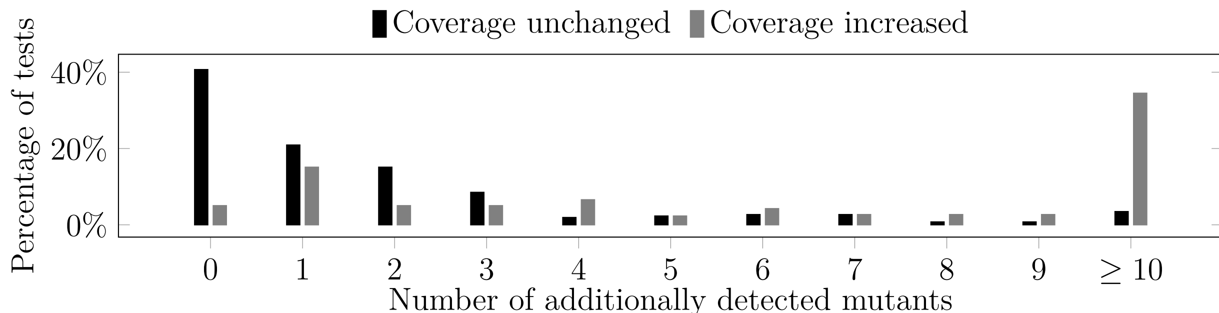


Figure 4.5: Effect of triggering tests on mutant detection.
The bars represent 480 triggering tests. 258 triggering tests did not increase statement coverage (unchanged), and each one detects 2 new mutants on average. 222 triggering tests increased statement coverage, and each one detects 28 new mutants on average.

analyses. In contrast, there was no significant association between *mutation score increased* and *test added*.

In addition to determining whether the mutation score increased, we also measured the sensitivity of the mutation score with respect to the detection of a single real fault, i.e., the number of mutants additionally detected by the triggering test. Figure 4.5 visualizes the number of additionally detected mutants when coverage did not increase (unchanged) and when it did. For triggering tests that did not increase statement coverage (black bars), two characteristics can be observed. First, 40% of these triggering tests did not detect any additional mutants. Second, 45% of these triggering tests detected only 1–3 additional mutants, suggesting that the number of mutants that are coupled to a real fault is small when accounting for the conflating effects of code coverage.

Figure 4.5 also illustrates these conflating effects of code coverage on the mutation score: 35% of triggering tests that increased statement coverage (gray bars) detected 10 or more additional mutants. In contrast, this ratio was only 3% for triggering tests that did not increase statement coverage.

We also investigated the underlying mutation operators of the mutants that are coupled to real faults when statement coverage did not increase. We found that real faults were more often coupled to mutants generated by the *conditional operator replacement*, *relational operator replacement*, and *statement deletion* mutation operators. A possible explanation is that some of these mutants cannot be detected by tests that only satisfy statement coverage. Conditional and relational operator replacement mutants are frequently generated within

conditional statements, and numerous statement deletion mutants only omit side effects – detecting those mutants requires more thorough testing. None of these three mutation operators is known to generate a disproportionate number of equivalent mutants [114], hence they should always be employed during mutation analysis.

> **Answer 4.1.** *73% of real faults are coupled to the mutants generated by commonly used mutation operators. When controlling for code coverage, on average 2 mutants are coupled to a single real fault, and the conditional operator replacement, relational operator replacement, and statement deletion mutants are more often coupled to real faults than other mutants.*

4.2.2 What Types of Real Faults Are Not Coupled to Mutants?

For 95 out of 357 real faults (27%), none of the triggering tests detected any additional mutants. We manually reviewed each such fault to investigate whether this indicates a general limitation of mutation analysis. Table 4.4 summarizes the results, which fell into three categories: cases where a mutation operator should be strengthened, cases where a new mutation operator should be introduced, and cases where no obvious mutation operator can generate mutants that are coupled to the real fault. In the latter case, results derived from mutation analysis do not generalize to those real faults.

Real faults requiring stronger mutation operators (25)

- *Statement deletion (12)*: The statement deletion operator is usually not implemented for statements that manipulate the control flow. We surmise that this is due to technical challenges in the context of Java — removing `return` or `break/continue` statements changes the control flow and may lead to uninitialized variables or unreachable code errors. Figure 4.6a gives an example.
- *Argument swapping (6)*: Arguments to a method call that have the same type can be swapped without causing type-checking errors. Argument swapping represents a special case of swapping identifiers, which is not a commonly-used mutation operator [83]. Figure 4.6b shows an example.
- *Argument omission (5)*: Method overloading is error-prone when two methods differ in one extra argument — a developer might inadvertently call the method that requires fewer arguments. Figure 4.6c gives an example. Generating mutants for this type

Table 4.4: Number of real faults not coupled to mutants generated by commonly used mutation operators. Numbers are categorized by reason: weak implementation of a mutation operator, missing mutation operator, or no appropriate mutation operator exists.

	Weak operator	Missing operator	No such operator	Total
Chart	5	1	2	8
Closure	11	2	18	31
Math	4	4	30	38
Time	2	0	5	7
Lang	3	0	8	11
Total	25	7	63	95

of fault requires a generalization of a suggested class-based mutation operator, which addresses method overloading to a certain extent [58].

- *Similar library method called (2)*: Existing mutation operators replace one method call by another only for calls to getter and setter methods. It would be both unfeasible and largely unproductive to replace every method call with every possible alternative that type-checks. Nonetheless, the method call replacement operator should be extended to substitute calls to methods with related semantics — in particular library method calls for string operations. Figure 4.6d shows an example in which the fault is caused by using the wrong one of two similar library methods (`indexOf` instead of `lastIndexOf`).

Real faults requiring new mutation operators (7)

- *Omit chaining method call (4)*: A developer might forget to call a method whose return type is equal to (or a subtype of) its argument type. Figure 4.6e gives an example in which a string needs to be escaped. A new mutation operator could replace such a method call with its argument, provided that the mutated code type-checks.
- *Direct access of field (2)*: When a class includes non-trivial getter or setter methods for a field (e.g., further side effects or post-processing), an object that accesses the field directly might cause an error. Figure 4.6f shows an example in which post-processing of the field `chromosomes` is required before the method `iterator()` should be invoked. A new mutation operator could replace calls to non-trivial getter and setter methods with a direct access to the field.

```
    }
+   return false;
  }
  case 4: {
    char ch = str.charAt(0);
```

(a) Lang-365 fix

```
- Partial newPartial = new Partial(iChronology, newTypes,
   newValues);
+ Partial newPartial = new Partial(newTypes, newValues,
   iChronology);
```

(b) Time-88 fix

```
- return solve(min, max);
+ return solve(f, min, max);
```

(c) Math-369 fix

```
- int indexOfDot = namespace.indexOf('.');
+ int indexOfDot = namespace.lastIndexOf('.');
```

(d) Closure-747 fix

```
- return ... + toolTipText + ...;
+ return ... + ImageMapUtilities.htmlEscape(toolTipText) + ...;
```

(e) Chart-591 fix

```
- return chromosomes.iterator();
+ return getChromosomes().iterator();
```

(f) Math-779 fix

```
- FastMath.pow(2 * FastMath.PI, -dim / 2)
+ FastMath.pow(2 * FastMath.PI, -0.5 * dim)
```

(g) Math-929 fix

Figure 4.6: Snippets of real faults that require stronger or new mutation operators.

- *Type conversions (1)*: Wrong assumptions about implicit type conversions and missing casts in arithmetic expressions can cause unexpected behavior. Figure 4.6g shows an example where the division should be performed on floating point numbers rather than integers (the replacement of the division by multiplication is unrelated to the real fault). A new mutation operator could replace a floating-point constant by an exact integer equivalent (e.g., replace `2.0` by `2`), remove explicit casts, or manipulate operator precedence.

Real faults not coupled to mutants (63)

For the faults in this group, no obvious mutation operator exists that can simulate the faults. Note that it may be possible to simulate a particular instance of the fault, such as the examples we discuss, but that the class of faults as a whole cannot be simulated by a mutation operator.

- *Algorithm modification or simplification (37)*: Most of the real faults not coupled to mutants were due to incorrect algorithms. The bug fix was to re-implement or modify the algorithm.
- *Code deletion (7)*: Faults caused by extra code that has to be deleted are not coupled to mutants. A bug fix that only removes special handling code also falls into this category — Figure 4.7a gives an example.
- *Similar method called (5)*: Another common mistake is calling a wrong but related method within the program, which might either return wrong data or omit side-effects. Figure 4.7b shows an example of calling a wrong method. Note that this type of fault can be represented by mutants for well-known library methods. However, without deeper knowledge about the relation between methods in a program, replacing every identifier and method call with all alternatives would result in an unmanageable number of mutants.
- *Context sensitivity (4)*: Mutation analysis is context insensitive, while bugs can be context sensitive. Suppose the access of a field that might be null is extracted to a utility method that includes a null check. A developer might forget to replace an instance of the field access with a call to this utility method. This rather subtle fault cannot be represented with mutants since it would require inlining the utility method (without the null check) for every call. Figure 4.7c gives an example of this type of fault. The fault is that `this.startData` might be null — this condition is checked in `getCategoryCount()`. However, other tests directly or indirectly detect all mutants in `getCategoryCount()`, hence a test that exposes the fault does not detect any additional mutants.

- *Violation of pre/post conditions or invariants (3)*: Some real faults were caused by the misuse of libraries. For example, the Java library makes assumptions about the `hashCode` and `equals` methods of objects that are used as keys to a `HashMap`. Yet, a violation of this assumption cannot be generally simulated with mutants. Figure 4.7d gives an example of such a fault.
- *Numerical analysis errors (4)*: Real faults caused by overflows, underflows, and improper handling of NaN values are difficult to simulate with mutants, and hence also represent a general limitation. Figure 4.7e shows an example of a non-trivial case of this type of fault.
- *Specific literal replacements (3)*: Literal replacement is a commonly used mutation operator that replaces a literal with a well-defined default (e.g., an integer with 0 or a string with the empty string). However, the real fault might only be exposed with a specific replacement. For example, a map might contain a wrong value that is only accessible with a specific key. The literal replacement operator cannot generally simulate such a specific replacement. Figure 4.7f demonstrates an example that involves Unicode characters.

Answer 4.2. *27% of real faults are not coupled to the mutants generated by commonly used mutation operators. The set of commonly used mutation operators should be enhanced. However, 17% of real faults, mostly involving algorithmic changes or code deletion, are not coupled to any mutants.*

4.2.3 Is Mutant Detection Correlated with Real Fault Detection?

Section 4.2.1 provided evidence that mutants and real faults are coupled, but the question remains whether a test suite's mutation score is correlated with its real fault detection rate and whether mutation score is a good predictor of fault-finding effectiveness.

Figure 4.8 summarizes the point-biserial and rank-biserial correlation coefficients between the mutation score and real fault detection rate for each subject program. Both correlation coefficients lead to the same conclusion: the correlation is positive, usually strong or moderate, indicating that mutation score is indeed correlated with real fault detection. Unsurprisingly, real faults that are not coupled to mutants show a negligible or even negative correlation. For reference Figure 4.8 also includes the results for statement coverage.

The correlation between mutation score and real fault detection rate is conflated with the influence of statement coverage, but the Wilcoxon signed-rank test showed that the correlation coefficient between mutation score and real fault detection rate is significantly

```
  if (childType.isDict()) {
    ...
- } else if (n.getJSType != null &&
-     parent.isAssign()) {
-   return;
  } ...
```

(a) Closure-810 fix

```
- return getPct((Comparable<?>) v);
+ return getCumPct((Comparable<?>) v);
```

(b) Math-337 fix

```
- if (categoryKeys.length != this.startData[0].length)
+ if (categoryKeys.length != getCategoryCount())
```

(c) Chart-834 fix

```
- lookupMap = new HashMap<CharSequence,CharSequence>();
+ lookupMap = new HashMap<String,CharSequence>();
```

(d) Lang-882 fix[a]

[a] The result of comparing two `CharSequence` objects is undefined — the bug fix uses `String` to alleviate this issue.

```
- if (u * v == 0)
+ if ((u == 0) || (v == 0))
```

(e) Math-238 fix

```
- {"\u00CB", "&Ecirc;"},
+ {"\u00CA", "&Ecirc;"},
+ {"\u00CB", "&Euml;"},
```

(f) Lang-658 fix

Figure 4.7: Snippets of real faults not coupled to mutants.

higher than the correlation coefficient between statement coverage and real fault detection rate for all subject programs except Time.

Further investigating the influence of statement coverage, Table 4.5 summarizes the comparison of the mutation scores between all test suites $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$ for each real fault when coverage is controlled or ignored (not controlled). In addition to the number of real faults for which the mutation score of $\tilde{T}_{fail}$ is significantly higher, the table shows the average $\hat{A}_{12}$ effect size.

Table 4.5: Comparison of mutation scores between $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$.

Significant gives the number of real faults for which $\tilde{T}_{fail}$ has a significantly higher mutation score (Wilcoxon signed-rank test, significance level 0.05).

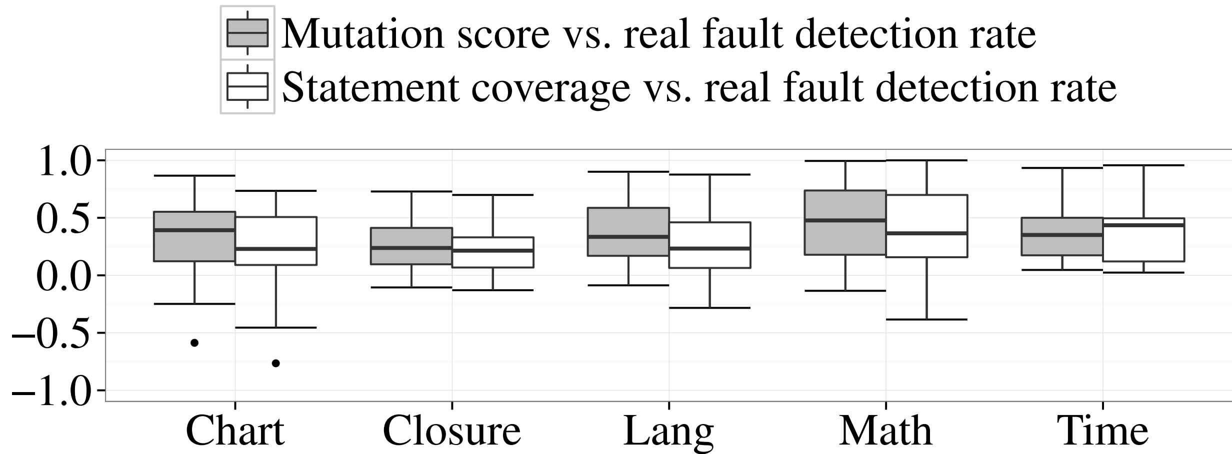
Program	**Coverage controlled**		**Coverage ignored**	
	Significant	Avg. $\hat{A}_{12}$	Significant	Avg. $\hat{A}_{12}$
Chart	21/22	0.74	21/22	0.74
Closure	27/32	0.66	30/32	0.71
Math	76/80	0.79	80/80	0.81
Time	18/18	0.81	17/18	0.81
Lang	40/42	0.77	39/42	0.78

Figure 4.9 summarizes the $\hat{A}_{12}$ effect sizes. In our scenario the value of $\hat{A}_{12}$ is an estimation of the probability that a test suite with a higher real fault detection rate has a higher mutation score as well, where a value $\hat{A}_{12} = 1$ means that the mutation score increased for all observations. An effect size of $\hat{A}_{12} \geq 0.71$ is typically interpreted as large. As expected, the effect size is greater if statement coverage is ignored (not controlled), but the average effect size remains large for all subject programs except for Closure when coverage is controlled.

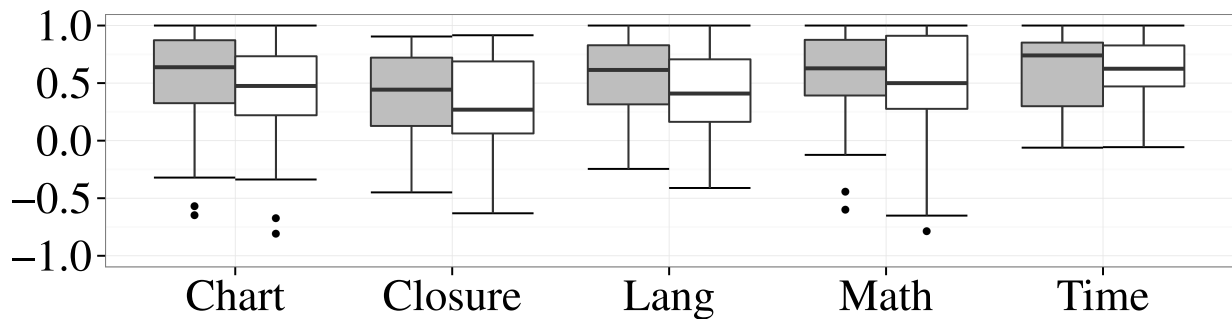
Answer 4.3. *Mutant detection is positively correlated with real fault detection, independently of code coverage. This correlation is stronger than the correlation between statement coverage and real fault detection.*

4.2.4 Threats to Validity

Our evaluation uses only 5 subject programs, all written in Java. Other programs might have different characteristics. Moreover, all 5 subject programs are well-tested (see Figure 4.3). This may limit the applicability of the results to programs that are not well-tested (e.g., programs under development). However, we do not feel this is a major threat, since mutation analysis is typically only used as an advanced metric. For example, if a test suite covers only 20% of the source code, developers are likely to focus on improving code coverage before they focus on improving the mutation score.



(a) Point-biserial correlation coefficients.



(b) Rank-biserial correlation coefficients.

Figure 4.8: Correlation coefficients for each subject program.
The differences between the correlation coefficients (of mutation score and statement coverage) are significant (Wilcoxon signed-rank test) for all subject programs ($p < 0.05$) except Time ($p > 0.2$).

Another threat to validity is the possibility of a bias in our fault sample. We located the real faults by automatically linking bug identifiers from the bug tracking system to the version control revisions that resolved them. Previous work suggests that this approach does not produce an unbiased sample of real faults [10]. In particular, the authors found that not all faults are mentioned in the bug tracking system and that not all bug-fixing commits can be identified automatically. In addition, they found that process metrics such as the experience of the developer affect the likelihood that a link will be created between the issue and the commit that fixes it. However, this threat is unlikely to impact our results for the following two reasons. First, while we may suffer false negatives (i.e., missed faults),

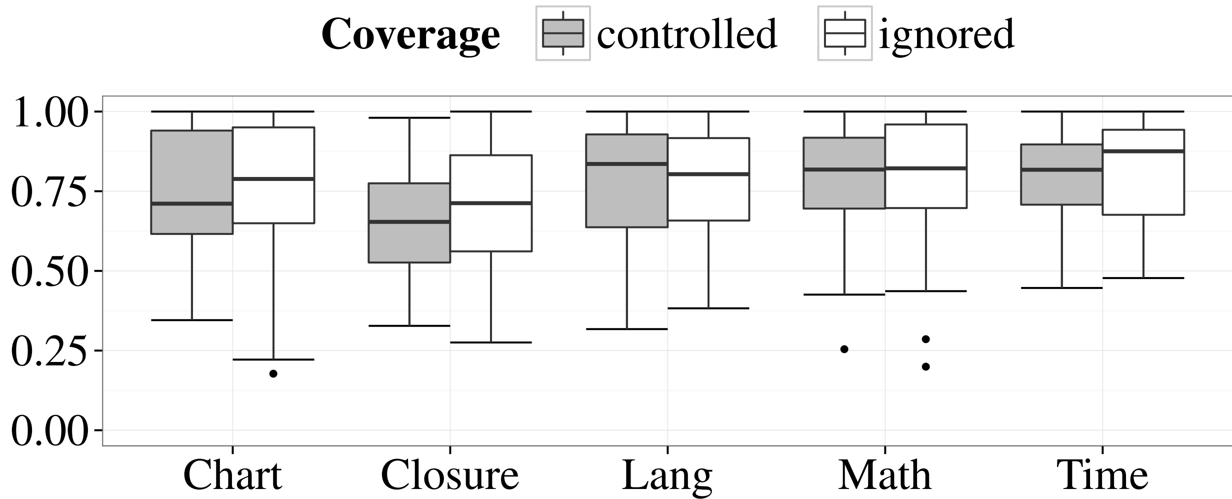


Figure 4.9: $\hat{A}_{12}$ effect sizes for mutation score differences between $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$ for each subject program.

our dataset is unlikely to be skewed towards certain *types* of faults, such as off-by-one errors. Bachmann et al. did not find a relationship between the type of a fault and the likelihood that the fault is linked to a commit [10]. Second, recent evidence suggests that the size of bug datasets influences the accuracy of research studies more than the bias of bug datasets [93]. The severity of the bias threat is therefore reduced by the fact that we used a large number of real faults in our study.

We focused on identifying faults that have an unquestionably undesirable effect and that can be triggered with an automated test. It is possible that our results – the correlation between the mutant detection and real fault detection – do not generalize to faults that do not match these criteria. However, we argue that reproducibility of faults is desirable and characteristic of common practice.

A final threat is that we did not use class-level mutation operators, such as those in Kim and Offutt's studies [58, 84]. We did not consider them in our study for two reasons. First, class-level mutation operators are neither implemented in modern Java mutation tools such as Major, Javalanche, and PIT, nor are they commonly used in experiments involving mutants. We therefore argue that using the set of traditional mutation operators improves comparability and also generalizability – the set of traditional mutation operators is applicable to many programming languages. In addition, our qualitative study addresses this threat and shows whether and how mutation analysis could benefit from adding improved or specialized versions of class-level mutation operators.

Table 4.6: Comparison of studies that explored the relationship between mutants and real faults.

LOC gives the total number of lines of code of the studied programs that contained real faults. Test suites gives the type of used test suites (*gen*=generated, *dev*=developer-written). Mutation operators refers to: *Rc*=Replace constants, *Ri*=Replace identifiers, *Ro*=Replace operators, *Nbc*=Negate branch conditions, *Ds*=Delete statements, *Mbc*=Modify branch conditions (note that *Mbc* subsumes *Nbc* [54]).

	Real faults	LOC	Tests suites	Mutation operators	Mutants evaluated	Coverage controlled
[19]	12	1,000	*gen*	*Rc*, *Ri*, *Ro*	1%	no
[6]	38	5,905	*gen*	*Rc*, *Ro*, *Nbc*, *Ds*	10%	no
[79]	38	5,905	*gen*	*Rc*, *Ri*, *Ro*, *Nbc*, *Ds*	10%	no
Our study	357	321,000	*gen* *dev*	*Rc*, *Ro*, *Mbc*, *Ds*	100%	yes

4.3 Related Work

This section discusses previous studies that explored the relationship between mutants and real faults. It also discusses commonly used artifacts that provide faulty program versions and research areas that rely on the existence of a correlation between the mutation score and real fault detection rate.

4.3.1 Studies That Explored the Relationship Between Mutants and Real Faults

We are only aware of three previous studies that investigated the relationship between mutants and real faults, which are summarized in Table 4.6.

Duran and Thévenod-Fosse [19] performed the first such study. They found that, when their subject program was exercised with generated test suites, the errors (incorrect internal state) and failures (incorrect output) produced by mutants were similar to those produced by real faults. However, this study was limited in scope as it considered a single 1,000 line C program and evaluated only 1% of the generated mutants. Finally, this study only used generated test suites and did not control for code coverage.

Andrews et al. [6] were the next to explore the relationship between mutants, hand-seeded faults, and real faults. They found that hand-seeded faults are not a good substitute for real faults, but that mutants are. In particular, they found that there is no practically significant difference between the mutation score and the real fault detection rate. However, this study was also limited in scope since only one of the eight studied programs (Space) contained real faults. Space is written in C and contains 5,905 lines of code. Additionally, the study considered only 10% of the generated mutants, used automatically-generated test cases, and did not control for code coverage.

Namin and Kakarla [79] later replicated the work of Andrews et al. [6], used a different mutation testing tool (Proteum), and came to a different conclusion: they found that the correlation between the mutation score and the real fault detection rate for Space was weak. They also extended the work to five Java classes from the standard library, ranging from 197 to 895 lines of code. Faults were hand-seeded by graduate students, and the authors found that the correlation was considerably stronger.

To the best of our knowledge, our study is the first to undertake experimental evaluation of the relationship between mutants and real faults at such a scale in terms of number of real faults, number of mutants, subject program size, subject program diversity, and the use of developer-written and automatically-generated test suites. In addition, our study is the first to consider the conflating effects of code coverage on the mutation score and the first to explore real faults in object-oriented programs.

4.3.2 Commonly Used Artifacts

Many research papers use programs from the Siemens benchmark suite [42] or the software-artifact infrastructure repository (SIR) [21] in their evaluation. More precisely, Google Scholar lists approximately 1,400 papers that used programs from the Siemens benchmark suite, and SIR's usage information website [104] lists more than 500 papers that reference SIR.

The Siemens benchmark suite consists of 7 C programs varying between 141 and 512 lines of code, and all faults were manually seeded. The authors described their manually-seeded faults as follows [42]: "The faults are mostly changes to single lines of code, but a few involve multiple changes. Many of the faults take the form of simple mutations or missing code." Thus, our results likely hold for these faults, which are essentially mutants.

SIR provides 81 subjects written in Java, C, C++, and C#. According to the SIR meta data, 36 of these subjects come with real faults. The median size of those 36 subjects is 120 lines of code, and 35 of them are written in Java. SIR was not suitable for our study

due to the small program sizes and the absence of comprehensive developer-written test suites. Therefore, we developed a fault database that provides 357 real faults for 5 large open-source programs, which feature comprehensive test suites [52].

4.3.3 Software Testing Research Using Mutants

The assumption that mutant detection is well correlated with real fault detection underpins many studies and techniques in several areas in software testing research.

Mutation analysis is an integral part of mutation-based test generation approaches, which automatically generate tests that can distinguish mutant versions of a program from the original version (e.g., [25, 37, 87, 118]). However, studies in this area have not evaluated whether the generated test suites can detect real faults.

Test suite minimization and prioritization approaches are often evaluated with mutants to ensure that they do not decrease (or they minimally decrease) the mutation score of the test suite (e.g., [22, 95]). Prior studies, however, left open the question whether and how well those approaches maintain real fault effectiveness.

To evaluate an algorithm for fault localization or automatic program repair, one must know where the faults in the program are. Mutants are valuable for this reason and commonly used (e.g., [15, 49]). Yet, it is unclear whether those algorithms evaluated on mutants perform equally well on real faults.

Our qualitative and quantitative studies show to what extent research using mutants generalizes to real faults. Our studies also reveal inherent limitations of mutation analysis that should be kept in mind when drawing conclusions based on mutants.

4.4 Relevance to Thesis

Recall that our thesis statement is as follows:

Thesis Statement. *Robust development processes are necessary to minimize the number of faults introduced when evolving complex software systems. These processes should be based on empirical research findings. Data science techniques allow software engineering researchers to develop research insights that may be difficult or impossible to obtain with other research methodologies. These research insights support the creation of development processes. Thus, data science techniques support the creation of empirically-based development processes.*

Testing is a powerful quality assurance tool. However, there are many different approaches to testing software. Comparing these approaches to see which is the most cost-effective requires doing testing research; conducting testing research requires having faults to detect. Unfortunately, the real faults that remain in "wild" software are by definition unknown. Testing researchers therefore require some way of introducing faults into software in order to determine if a given test suite can detect those faults. Mutants are a popular choice for this as mutation testing provides an automated, replicable way of seeding faults. However, prior to this work, empirical support for the use of mutants was lacking. Our work shows that a test suite's mutant kill score is correlated with its ability to detect real faults, supporting the use of mutants in testing research. The work also shows that mutation testing provides a good estimate of a suite's fault detection ability and therefore can be recommended to practitioners who are interested in measuring the quality of their test suites.

To test for the existence of the correlation between mutant detection and fault detection, we had to identify real faults in real programs. Though this required some manual effort, the use of data science techniques was essential to measure this correlation. Generating test suites, in particular, took over a year of serial machine time and could not have been done without access to multiple computational clusters. Thus, this chapter of the thesis provides additional evidence for our thesis statement: data science tools permit the development of new research insights that in turn affect the day-to-day practice of software engineering.

Chapter 5

Conclusion

> *Driving decisions with data is like having a spotlight. It creates an area of clarity and illumination but also shadows and dark spots.*
>
> *Kent Beck, 2016 [12]*

The scope and complexity of modern software systems makes it difficult for developers to fully understand the ramifications of a given code change. For this reason, software development companies create processes for developers to follow when maintaining and evolving code. The use of good processes can reduce the number of faults introduced into the system. Naturally, the challenge is defining "good"; this is where software engineering researchers come in.

Software engineering researchers develop many insights that can help developers perform their day-to-day work. There are many different methodologies that can be used to conduct this research, including interviews, surveys, and industrial case studies. Each approach has some benefits and some drawbacks when compared to the others. In this thesis, we explored how the use of data science techniques can generate research insights that could not be obtained in other ways. We saw three examples that support this claim.

In Chapter 2, we saw that mobile application (app) research uses app corpora of varying sizes, ranging from tens of apps to hundreds of thousands of apps. This makes it hard to judge the generalizability of any one study. We therefore developed empirical guidelines that can assist researchers studying mobile apps. We suggested that a corpus containing 1,000 apps may be sufficient if the research is concerned with typical behaviour, while a million or more apps may be required if the study seeks to identify outliers. We also showed that 13% of the public API methods were not used by the apps in our corpus of 1,368,376. This

information can help guide the Android maintainers as they choose methods to deprecate. In addition, it will indirectly benefit users of the Android API who will have a smaller API to comprehend.

Next, we applied our newly-created guideline to a study of Android malware detection. Previous studies of Android malware detection did not control for the API level of the apps in the corpus and none used more than 135,792 apps. This represented a potential threat to the validity of at least fifteen previous studies on the topic. We conducted a study of 1.3 million apps in which we controlled for API level. As it happened, the potential threat to validity did not materialize: the accuracy of the malware detection classifiers is not artificially inflated when API level is not controlled. However, this information helps researchers working with Android by increasing their confidence in existing results. Future work on this topic could explore the cause of this result, as it is somewhat counterintuitive.

In Chapter 3, we saw that the regression test suites we studied rarely detected a fault: only 0.28% of test case executions failed due to a fault in the system under test. In addition, 26% of non-flaky test failures were resolved by fixing the test, not the system under test. The ideal tradeoff between the cost of production bugs and the cost of writing, maintaining, and executing a test suite naturally depends on the software system in question. However, while developers are intellectually aware that every test has a cost, they likely do not consider the lifetime cost of a test as they are writing it. The findings we presented in this chapter may make them more aware of the cost of regression testing relative to the benefits, informing their test suite maintenance decisions. Future work in this area could broaden the study to other platforms and languages as well as replicating the results at individual companies in the form of in-depth case studies.

In Chapter 4, we saw that a great deal of testing research relies on the assumption that mutants are an adequate substitute for real faults. That is, the studies assume that a test suite's mutant detection rate is correlated with its real fault detection rate. However, prior to our study, this assumption was not well supported by empirical evidence. We showed that the mutant detection rate and the real fault detection rate are indeed correlated, even when the effect of code coverage is accounted for. Future work in this area could explore whether equivalent mutants are indicative of code smells.

Taken together, these three studies provide examples of using data science to generate research results that can inform development processes. The results in Chapter 3 are directly applicable to developers, while the results in Chapters 2 and 4 affect software engineering research methodology and thus indirectly impact developers. Data science techniques played a key role in the method used to conduct all three studies. Without these techniques, we would have been unable to handle a corpus of 1.3 million apps, study test

suites for 61 projects, or identify 437 faults to compare to mutant programs. Of course, as the quote at the beginning of this chapter highlights, facts and figures extracted from a software system cannot describe it completely. In addition, the data in use may be incomplete, noisy, biased, unstructured, or of uncertain provenance. Finally, if the data are generated by system users, the use of data science may raise privacy and security issues. Thus, data science is not a panacea for attacking software development problems. Human judgement is needed to interpret and apply the information produced by data science. However, when used carefully, research results generated via the application of data science techniques can replace the use of heuristics and intuition, making the maintenance of large, complex software systems much easier.

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Proceedings of the International Conference on Security and Privacy in Communication Networks*, 2013.

[2] Kevin Allix, Tegawendé Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Large-scale machine learning-based malware detection: Confronting the 10-fold cross validation scheme with reality. In *Proceedings of the Conference on Data and Application Security and Privacy*, 2014.

[3] Kevin Allix, Tegawendé Bissyande, Jacques Klein, and Yves Le Traon. Machine learning-based malware detection for Android applications: History matters! Technical report, University of Luxembourg, SnT, 2014.

[4] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering*, 21(1), 2014.

[5] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the Symposium on the Foundations of Computer Science*, 2006.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[8] A. M. Aswini and P. Vinod. Droid permission miner: Mining prominent permissions for Android malware analysis. In *Proceedings of the International Conference on the Applications of Digital Information and Web Technologies*, 2014.

[9] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[10] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 97–106, 2010.

[11] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[12] Kent Beck. Tweet details, 2016.

[13] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.

[14] Daniel Brahneborg. #521: [PATCH] Avoid uninitialized data in random buffer.

[15] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 782–791, 2013.

[16] Cobertura. The official web site of the Cobertura project, Accessed Jan 28, 2014. `http://cobertura.sourceforge.net`.

[17] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[18] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254, 2006.

[19] Muriel Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1996.

[20] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 4(11):34–41, 1978.

[21] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering (ESEM)*, 10(4):405–435, 2005.

[22] Sebastian Elbaum, Alexy G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering (TSE)*, 28(2):159–182, 2002.

[23] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[24] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[25] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[26] Rudolf Freund, William Wilson, and Donna Mohr. *Statistical Methods*, pages 151–153. Academic Press, 2010.

[27] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware. Technical Report GMU-CS-TR-2015-10, George Mason University, 2015.

[28] Gartner. Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015, 2016.

[29] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of the Workshop on Artificial Intelligence and Security*, 2013.

[30] William Glodek and Richard Harang. Rapid permissions-based detection and analysis of mobile malware using random decision forests. In *Proceedings of the Military Communications Conference*, 2013.

[31] Google. Linus torvalds on git, 2007.

[32] Google. Android security 2014 year in review, 2014.

[33] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *Transactions on Software Engineering and Methodology*, 10(2), 2001.

[34] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[35] Latifa Guerrouj, Shams Azad, and Peter C. Rigby. The influence of app churn on app success and StackOverflow discussions. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, 2015.

[36] David Heinemeier Hansson. Testing like the TSA. https://signalvnoise.com/posts/3159-testing-like-the-tsa.

[37] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 212–222. ACM, 2011.

[38] Ahmed Hassan. *Mining software repositories to assist developers and support managers.* PhD thesis, University of Waterloo, 2004.

[39] Michi Henning. API design matters. *Queue*, 5(4), 2007.

[40] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[41] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[42] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.

[43] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.

[44] Laura Inozemtseva and Reid Holmes. Using machine learning to detect android malware: Should classifier evaluations control app API level?, 2017. Submitted to ICSME 2017.

[45] Laura Inozemtseva, Subramanian Siddharth, and Reid Holmes. Integrating software project resources using source code identifiers. In *Proceedings of the International Conference on Software Engineering*, 2014. Accepted to the New Ideas and Emerging Results track.

[46] Saeed Salem Jeff Anderson and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the Working Conference on Mining Software Repositories*, 2014.

[47] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[48] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2011.

[49] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[50] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, 2014.

[51] René Just, Michael D. Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 315–326, 2014.

[52] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[53] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[54] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 720–725, 2012.

[55] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[56] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: Empirical observations. *Advances in Software Engineering*, 2010.

[57] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar Devanbu, and Vladimir Filkov. Using and asking: APIs used in the Android market and asked about in StackOverflow. In *Proceedings of the Conference on Social Informatics*, 2013.

[58] Sunwoo Kim, John A Clark, and John A McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems*, pages 9–12, 2000.

[59] Negar Koochakzadeh and Vahid Garousi. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering*, 2010.

[60] Negar Koochakzadeh, Vahid Garousi, and Frank Maurer. Test redundancy measurement based on coverage information: Evaluations and lessons learned. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2009.

[61] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration, 2017. Submitted to FSE 2017.

[62] Yash Lamba, Manisha Khattar, and Ashish Sureka. Pravaaha: Mining Android applications for discovering API call usage patterns and trends. In *Proceedings of the India Software Engineering Conference*, 2015.

[63] Nancy Leveson and Clark Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7), 1993.

[64] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential component leaks in Android apps: An investigation into a new feature set for malware detection. In *Proceedings of the International Conference on Software Quality, Reliability and Security*, 2015.

[65] Li Li, Tegawendé Bissyandé, Jacques Klein, and Yves Le Traon. Parameter values of Android APIs: A preliminary study on 100,000 apps. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[66] Li Li, Tegawendé Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible Android APIs: An empirical study. In *Proceedings of the International Conference on Software Maintenance and Evolution*, 2016.

[67] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Proceedings of the Working Conference on Mining Software Repositories*, 2014.

[68] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger Stack Overflow discussions? A study on the Android SDK. In *Proceedings of the International Conference on Program Comprehension*, 2014.

[69] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: A decade of wasted cores. In *Proceedings of the European Conference on Computer Systems*, 2016.

[70] Cosmin Marsavina, Daniela Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, 2014.

[71] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *Proceedings of the Working Conference on Mining Software Repositories*, 2015.

[72] McAfee. McAfee labs threats reports, 2014.

[73] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the International Conference on Software Maintenance*, 2013.

[74] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2003.

[75] Cade Metz. Google mistakes entire web for malware, 2009.

[76] Trend Micro. Mind the (security) gaps: The 1H 2015 mobile threat landscape, 2015.

[77] Zach Miners. Report: Malware-infected android apps spike in the google play store, 2014.

[78] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2013.

[79] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 342–352, 2011.

[80] A.S. Namin, J.H. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 351–360, 2008.

[81] Jon Oberheide and Charlie Miller. Dissecting the Android bouncer, 2012. `http://diyhpl.us/~bryan/papers2/security/android/summercon12-bouncer.pdf`.

[82] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[83] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[84] Jeff Offutt, Yu-Seng Ma, and Yong-Rae Kwon. The class-level mutants of MuJava. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 78–84, 2006.

[85] Thomas Ostrand, Elaine Weyuker, and Robert Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, 31(4):340–355, 2005.

[86] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[87] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.

[88] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical Report GIT-CS-12-05, Georgia Institute of Technology, 2012.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[90] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2012.

[91] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*, pages 620–628. Cambridge University Press, 1992.

[92] Mykola Protsenko and Tilo Müller. Android malware detection based on software complexity metrics. In *Proceedings of the International Conference on Trust, Privacy & Security in Digital Business*, 2014.

[93] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 147–157, 2013.

[94] David Rosenblum and Elaine Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *Transactions on Software Engineering (TSE)*, 23(3), 1997.

[95] 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 of the International Conference on Software Maintenance (ICSM)*, pages 34–43, 1998.

[96] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002.

[97] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)*, 27(10):929–948, 2001.

[98] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for Android app security analysis using machine learning. In *Proceedings of the Annual Computer Security Applications Conference*, 2015.

[99] Justin Sahs and Latifur Khan. A machine learning approach to Android malware detection. In *Proceedings of the European Intelligence and Security Informatics Conference*, 2012.

[100] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. PUMA: Permission usage to detect malware in Android. In *Proceedings of the International Conference on Computational Intelligence, Information Security, and Data Mining*, 2013.

[101] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G. Bringas, and Gonzalo Álvarez Marañón. MAMA: Manifest analysis for malware detection in Android. *Cybernetics and Systems*, 44(6-7), 2013.

[102] Borja Sanz, Igor Santos, Javier Nieves, Carlos Laorden, Inigo Alonso-Gonzalez, and Pablo G. Bringas. MADS: Malicious Android applications detection through string analysis. In *Proceedings of the International Conference on Network and System Security*, 2013.

[103] Bruce Schneier. Random number bug in Debian Linux, 2008.

[104] SIR: Software-artifact Infrastructure Repository. SIR usage information, Accessed Mar 4, 2014. `http://sir.unl.edu/portal/usage.php`.

[105] Ben H Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering (ESEM)*, 14(3):341–369, 2009.

[106] Sophos. Security threat report 2014, 2014.

[107] Kunal Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2008.

[108] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. ** updated version with corrections**. *arXiv preprint arXiv:1512.01862*, 2015.

[109] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of Google Play. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2014.

[110] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[111] Roger Whitney. 2.8a1.390 Missing Dialog classes, 2007.

[112] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and API calls tracing. In *Proceedings of the Asia Joint Conference on Information Security*, 2012.

[113] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[114] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 919–930, 2014.

[115] Suleiman Yerima, Sakir Sezer, and Gavin McWilliams. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security*, 8(1), 2014.

[116] Suleiman Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new Android malware detection approach using Bayesian classification. In *Proceedings of the International Conference on Advanced Information Networking and Applications*, 2013.

[117] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of the International Conference on Software Testing Verification, and Validation*, 2008.

[118] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[119] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396, 2014.

[120] Jing Zhou and Robert Walker. API deprecation: A retrospective analysis and detection method for code examples on the web. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[121] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the Symposium on Security and Privacy*, 2012.

[122] Thomas Zimmermann. *Changes and Bugs—Mining and Predicting Development Activities*. PhD thesis, Saarland University, May 2008.