Which Software Faults Are Tests Not Detecting?

Jean Petrić
Lancaster University
Lancaster, UK
j.petric@lancaster.ac.uk

Tracy Hall
Lancaster University
Lancaster, UK
tracy.hall@lancaster.ac.uk

David Bowes
Lancaster University
Lancaster, UK
d.h.bowes@lancaster.ac.uk

ABSTRACT

Context: Software testing plays an important role in assuring the reliability of systems. Assessing the efficacy of testing remains challenging with few established test effectiveness metrics. Those metrics that have been used (e.g. coverage and mutation analysis) have been criticised for insufficiently differentiating between the faults detected by tests. **Objective:** We investigate how effective tests are at detecting different types of faults and whether some types of fault evade tests more than others. Our aim is to suggest to developers specific ways in which their tests need to be improved to increase fault detection. **Method:** We investigate seven fault types and analyse how often each goes undetected in 10 open source systems. We statistically look for any relationship between the test set and faults. **Results:** Our results suggest that the fault detection rates of unit tests are relatively low, typically finding only about a half of all faults. In addition, conditional boundary and method call removals are less well detected by tests than other fault types. **Conclusions:** We conclude that the testing of these open source systems needs to be improved across the board. In addition, despite boundary cases being long known to attract faults, tests covering boundaries need particular improvement. Overall, we recommend that developers do not rely only on code coverage and mutation score to measure the effectiveness of their tests.

CCS CONCEPTS

• **Software creation and management** → *Software verification and validation*; • **Software defect analysis** → *Software testing and debugging*.

KEYWORDS

software testing, unit tests, test effectiveness

ACM Reference Format:
Jean Petrić, Tracy Hall, and David Bowes. 2020. Which Software Faults Are Tests Not Detecting?. In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3383219.3383236

1 INTRODUCTION

Software testing is important to ensure that faults are detected before system deployment. Testing is an expensive activity accounting for more than 50% of development costs [4]. It is essential that testing activities are effective. Typical measures of test effectiveness[1] are based either on coverage or mutation testing. The limitation of such measures is that high granularity information is generated about test effectiveness. This information tends to be one overall percentage (i.e. percentage lines exercised by tests or percentage seeded faults detected by tests).

In this work, we aim to go beyond the overall numbers and analyse the ability of test suites to detect seven different fault types. Previous studies suggest that different faults are predicted by different machine learning models [15, 27], therefore it is likely that different faults are detected by different tests. Our results should allow developers to improve the overall effectiveness of their tests.

Previous studies have mostly been focused on investigating the relationship between the traditional effectiveness measures (e.g. code coverage) and faults. However, the overall figures computed by those measures are insufficient to pinpoint how to improve the ability of tests to detect more faults. Other factors have also been investigated in terms of the ability of tests to uncover faults. Most notably, test suite size [19], number of assertions [42] and number of covered methods per test [31] have all been shown to have an effect on test effectiveness. The relationship between fault type and test effectiveness has not been adequately explored.

The aim of this study is to answer the following research questions:

RQ1. What is the overall effectiveness of tests to detect faults in 10 open source systems?
RQ2. Which of seven fault types are least and most often detected by tests in these 10 open source systems?
RQ3. How can tests be improved to ensure the seven fault types covered in this study are effectively detected?

We answer these research questions by systematically selecting 10 open source systems in which we seed seven different types of artificial faults. We use the mutation testing tool PIT to seed the faults, which has been commonly used previously to simulate real faults in systems [17, 20]. We then run the unit tests associated with each system and record the faults detected and not detected. Following the execution of the test suites, we perform statistical analyses to establish which faults are detected more often than others. Finally, we assess the relative impact of each fault type on test effectiveness and suggest ways in which developers can improve their tests.

Our overall contributions are:

1) There seems to be a relatively low fault detection rate in all 10 open source systems. For six systems, less than a half of all faults are detected.

[1]Test effectiveness is usually defined as the ratio between the number of faults detected by the tests and the total number of faults in the system.

2) The distribution of detected fault types varies with some fault types detected more frequently than others. Conditional boundary and method call removal faults go undetected more often than increment and return value faults. In particular, conditional boundary faults get under-detected in all 10 systems analysed.

3) We demonstrate the relative impact on fault detection rate for each fault type across the 10 open source systems. For example, the lowest performing tests have a 10 times lower detection rate (on average) for boundary faults compared to other fault types.

4) Our results suggest ways in which developers can improve their tests by designing test cases which more effectively target faults that are infrequently detected.

This paper is structured as follows. In the following section we provide a background to this study. In the third section we describe the methodology, which is followed by results and discussion in the fourth section. We then describe related work in the fifth section and subsequently report on the threats to validity in the sixth section. Finally, in the last section we present the conclusions of this study.

2 BACKGROUND

Substantial effort is devoted to developing and maintaining test suites [41]. Unlike production code, which is checked against a test suite, the assessment of test effectiveness can be more challenging. Many studies have predominantly used techniques such as coverage and mutation score as proxies to assess test effectiveness [13, 16]. Other studies have investigated static features of tests to devise good practices for producing effective unit tests [10]. However, many challenges to ensure highly effective test suites remain.

Code coverage is a wide-spread technique to assess test effectiveness. In its simplest form coverage checks which lines or statements of code are covered by tests. A more complex form, branch coverage, assesses different paths in code with conditional expressions. There has been a great amount of debate on whether coverage is a suitable measure for test effectiveness. A weak but significant correlation between code coverage and the number of faults was found by Ahmed et al. [2]. Mockus et al. demonstrated that an increase in coverage proportionally reduces the number of faults [25]. Other researchers have raised concerns about coverage as a measure of test effectiveness, reporting that once confounding variables (e.g. test suite size) are controlled coverage does not perform well [18, 19].

Mutation testing seems increasingly popular as a technique to assess test effectiveness [30]. Mutation testing is inspired by biological processes, whereby the original code is slightly altered (mutated) and tests are executed against those changes. Tests should ideally fail when executed against mutated code (in the mutation analysis jargon, it is said, they should "kill" the mutant). Modifications to code are typically made by a predefined set of mutation operators, where each creates a different type of fault. Table 1 lists the mutation operators used in this study. Even though mutation testing is more sophisticated compared to coverage, there is no consensus on whether mutants are a valid replacement for real faults. Whilst Andrews et al. and Just et al. find evidence that mutants are a suitable replacement for real faults [5, 20], other researchers raise doubts about their usefulness [16, 26]. Even if mutations do not represent real faults currently in the code, it would be concerning if tests did not expose such faults given that they could occur in the future.

Purushothaman et al. empirically demonstrated that 90% of post release faults are complex; faults that can only be fixed by modifying code in multiple places [34]. Mutation testing has attempted to match complex faults with the introduction of higher order mutants (HOM). HOM are constructed by combining mutations of two or more first order mutants [23]. The challenging barrier with HOM is the rapid growth of the space of possible mutations which further slows down the process of mutation testing. However, there have been promising advances in reducing the space of possible mutants. Most prominent attempts have used search based techniques in order to reduce the HOM space (e.g. [24]). To date HOM is not in widespread use by practitioners and the availability of tools is limited.

Despite substantial efforts to deploy industry-scale mutation testing, some practitioners have raised several concerns that are yet to be addressed. Petrović et al. of Google argue that establishing test effectiveness via mutation testing is expensive [33]. Petrović et al. reported that even when redundant and equivalent mutants had been removed, too many unproductive mutants remained. Redundant mutants are a subset of mutants that have the same semantics, whilst equivalent mutants have the same behaviour as the original code from which they are derived. According to Petrović et al., unproductive mutants are those which are not useful in practice [33]. Once unproductive mutants were removed Petrović et al [33] reported that developers' satisfaction with mutation testing drastically increased from 20% to 80%. In addition, developers reported "many perceived benefits of mutation testing, including stronger tests, more effective debugging, prevention of bugs, and improved code quality".

Petrović et al. further argued that the ultimate rationale of the developer is to make a test suite better, rather than to merely increase mutation score. In line with this view, Bowes et al. investigated the effectiveness of tests using metrics that capture various facets of testing [10]. Consequently, several studies have empirically demonstrated that some metrics, such as the number of asserts [42] and methods invoked [31] by a test, are highly associated with faults. In this work we analyse mutants to identify how tests can be improved to cover specific types of fault.

Mutation testing approximates test effectiveness using mutation score. Mutation score is the ratio between the number of killed mutants (seeded faults exposed by a test) and overall mutants. The values span from 0 to 1, where 0 indicates poor, whilst 1 indicates perfect effectiveness. A notable caveat with respect to mutation score is the equivalent problem. A proportion of generated mutants might behave in the same way as the original code, in which case the calculation of mutation score can be deflated. The detection of equivalent mutants is a challenging and ongoing area of research [29].

To compare how often different mutants get detected, it is important to account for the equivalent problem. Equivalent mutants can make one type of fault appear more often than it can be killed, deflating the mutation score and leading to incorrect conclusions. To mitigate the equivalent problem, in this study we use the PIT mutation testing tool, version 1.4.5. PIT employs various techniques to

Table 1: The list of mutation operators used in this study (see 3.3 for selection criteria)

Name	Mnemonic	Description
Conditional Boundary Mutator	CB	The conditional boundary mutator replaces the relational operators <, <=, >, >= with their boundary counterpart.
Increments Mutator	I	The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.
Invert Negatives Mutator	IN	The invert negatives mutator inverts negation of integer and floating point numbers.
Maths Mutator	M	The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation.
Negate Conditionals Mutator	NC	The negate conditionals mutator will mutate all conditionals found, e.g. == to !=.
Return Values Mutator	R	The return values mutator mutates the return values of method calls.
Void Method Call Mutator	V	The void method call mutator removes method calls to void methods.

reduce the number of equivalent mutants, such as avoiding method calls to common logging frameworks[2]. The recent comparison study by Kintis et al. demonstrated that PIT produces substantially fewer equivalent mutants in comparison to other popular mutation testing tools (e.g. MuJava) [21], providing more reliable mutation scores.

Apart from coverage and mutation testing, a few other prominent approaches to improve test effectiveness have been proposed. According to Garousi, substantial effort has been put into reducing smells in tests [41]. Bavota et al. report their study of over 20 systems to investigate the effect of test smells on test maintainability [8]. They found that some test smells pose a potential risk to test maintenance. Tufano et al. demonstrated that some test smells influence code smells to appear in production code [37]. Other studies have shown that certain test code patterns have the ability to detect faults with high precision [38]. Athanasiou et al. also demonstrated that the quality of test code has a negative impact on production code [7].

Previous studies suggest that there are multiple facets which impact test effectiveness. Relying on a single metric such as coverage is unlikely to capture the underlying issues that make tests less effective. It is important to look beyond the numbers and find ways to improve tests. Mutation testing is a promising tool as it allows the collection of test data on a large scale which would otherwise be impractical to obtain. In comparison to coverage and test smells, mutation information provides most insight into real faults. This insight allows actionable improvements to tests by developers.

3 METHODOLOGY

3.1 Research Questions

In this study we are focused on answering three research questions.

RQ1. What is the overall effectiveness of tests to detect faults in 10 open source systems?

Test effectiveness has traditionally been used in previous studies to estimate how good tests are in detecting faults. In this study, we also examine the overall effectiveness of unit tests associated with the 10 analysed open source systems. We use overall mutation score as a proxy for overall test effectiveness.

RQ2. Which of seven fault types are least and most often detected by tests in these 10 open source systems?

Our second research question is based on the assumption that the detection rate of different fault types is not uniformly distributed. If this is the case, it would be useful to know which faults seem to remain undetected more than others. We use mutation operators as a proxy for fault types.

RQ3. How can tests be improved to ensure the seven fault types covered in this study are effectively detected?

We propose actionable practices that testers can use to improve test effectiveness.

3.2 Datasets

In this study we used 10 open source systems. To select the systems, we used the openly available list of 5000 GitHub repositories provided by Borges and Valente [9]. To narrow down the search for suitable systems to use in our analysis, we applied the following criteria:

(1) The system is actively developed. We used the number of commits to check whether the projects are active and look for at least 50 commits in the last two years (2017/2018) to consider a project.
(2) The system is popular in the community. Stars and forks are used as a proxy to identify suitably popular systems. We considered any project where the number of stars and forks is over 100.
(3) Needs to be a Java desktop/server application. We used the PIT tool [11] which is designed for running mutation analysis on Java projects.
(4) The build system needs to be Maven. Our data collection tools were designed to work with the Maven build system.

[2]http://pitest.org/quickstart/basic_concepts/

Table 2: Demographic data of the datasets used in this study

Project	Version	KLOC	Classes	TestCases	Coverage%[1]	Stars	Forks	Contributors
junit4	4.12	17	229	1463	86.80	5622	2184	142
dropwizard	1.3.8	33	537	320	86.40	5339	2335	331
guice	4.2.2	48	635	848	81.90	4432	734	49
metrics	4.0.5	13	206	226	80.80	4583	1224	167
jsoup	1.11.3	18	149	738	79.80	3860	1173	69
zxing	3.3.3	43	287	386	78.80	11884	5977	93
druid	1.1.11	219	1225	1986	73.20	5239	2711	105
activiti	7.0.9	152	1594	2097	66.80	1803	1795	161
retrofit	2.5.0	11	216	706	48.50	18399	3823	125
webmagic	0.7.3	12	196	49	39.20	3854	2148	34

[1] Total Coverage Percentage calculated by Atlassian Clover

(5) Running tests should be straightforward using the 'mvn test' command or a description on how to execute tests should be provided. Some projects rely on external dependencies and services which were not readily accessible.
(6) We moved down the list of 5000 until we found first 10 projects that satisfy the above criteria. Running mutation testing is a computationally demanding task which is why we selected 10 projects.

Table 2 shows demographic information about the 10 datasets selected. To collect the information about faults which get and do not get detected for each dataset, we ran mutation testing individually for each test class that contains at least one test case. Results were exported to a time-stamped directory per test class in the *XML* format. To collect the information about individual mutation survival rates we aggregated the PIT results from the *XML* files.

3.3 Mutation Operators

Table 1 summarises the seven default mutation operators defined in PIT (i.e. fault types) which are used in this study. We use Conditional Boundary, Increments, Invert Negatives, Maths, Negate Conditionals, Return Values and Void Method Call operators. When applied to a system under test, Conditional Boundary replaces a relational operator with its boundary counterpart [11]. For example, the relational operator < is replaced with <=, but not with > or >=. Similar applies to the Maths operator, where the original operator is replaced with its direct counterpart, for example the '+' sign is replaced with '−'. For more details on the other operators see [1].

The mutation operators we applied are the default operators in PIT [11]. We chose the default mutation operators as they are designed to "not be easy to detect" and "minimise the number of equivalent mutations that they generate" [1]. This is important for our study as we wanted to compare the survival rates of different mutation operators and avoid issues caused by equivalent mutants (i.e. mutants that behave the same as the original code).

3.4 Mutation Score

In this study we used the following equation to calculate mutation scores:

$$mScore = \frac{mutants_{killed}}{mutants_{killed} + mutants_{survived}} \quad (1)$$

Where generated mutants are not possible to kill (i.e. equivalent mutants), the use of $mutants_{survived}$ in Equation 1 may potentially deflate mutation scores. It is possible to mitigate this issue by replacing $mutants_{survived}$ with $mutants_{non-equivalent}$. However, the detection of equivalent mutants is a challenging task [29] which is typically not readily available in mutation testing tools (e.g. PIT [11]). Fortunately, PIT by default mitigates some of the issues caused by equivalent mutants as described in Section 2. In addition, since we did not favour any mutation operator, mutation scores were calculated across the operators and datasets consistently.

3.5 Analysis

To answer our research questions required multiple analysis steps. First, we calculated the overall mutation scores for each dataset. We then analysed the mutation scores of each individual mutation operator. The mutation scores of the individual mutation operators were calculated in a similar way to the overall scores. We grouped the statuses (i.e. killed or survived) of each fault by mutation operator. For each mutation operator we then calculated mutation scores according to Equation 1. To compare the mutation scores of different mutation operators we used a box plot and the Kruskal-Wallis non-parametric test. We set the significance level to 95%. We chose the Kruskal-Wallis test as it allows the comparison of 3 or more groups. In addition to the Kruskal-Wallis test, we used the Bonferroni correction to adjust for p-values involving multiple groups. The non-parametric test was chosen as the data was not normally distributed.

To assess the magnitude of the differences in mutation scores, we used two different approaches. First, we presented the relative distances of an individual mutation operator's score from the overall mutation operators' mean for each dataset. Relative distances depict the magnitude by how far each mutation operator is away from the overall mean. Negative relative distances suggest that more faults remain undetected by the tests, and vice-versa. Where a mutation operator distances itself from the mean, tests are either not robust enough to catch a fault, or perform better than the average. For our second approach, we calculated the mean values of each mutation operator across all datasets. This approach shows how the mutation scores of each mutation operator are distributed across the datasets. To present the relative magnitudes of mutation scores we used the logarithmic scale [39].

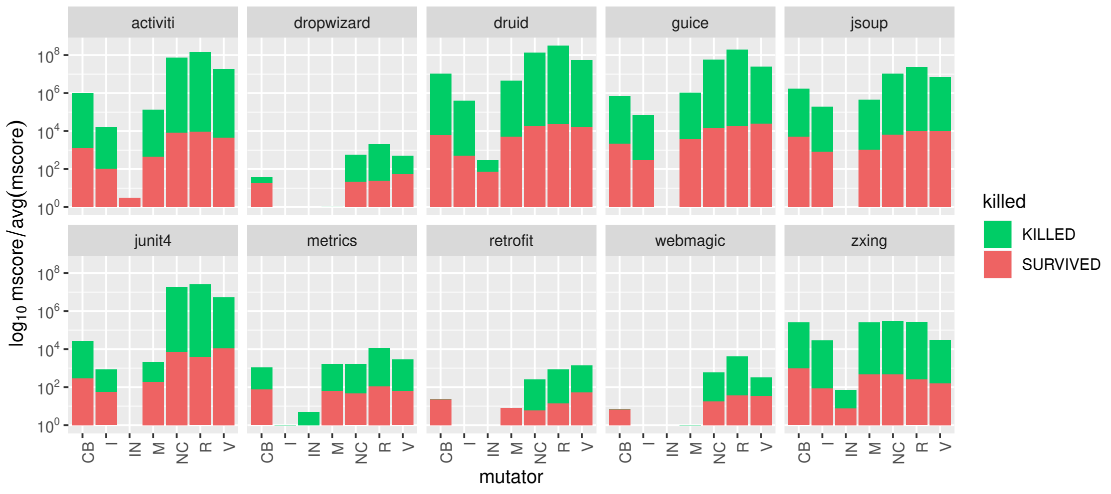

Figure 1: The frequency of killed and survived mutants per type and dataset

3.6 Experiment

The experiment was carried out as follows. We first compiled the source and test code of all projects reported in Table 2. We then identified all classes in all Java's test packages. To filter only test classes, we searched for test cases within the classes which used the @Test annotation placed before a test case. Tests using JUnit prior to 4.0 where identified by looking for methods with *test* in their method names. We manually searched for test classes as we experienced difficulties in running PIT on large multi-modal projects. The output of the first stage was a list of test classes to be fed into the PIT tool.

In the second stage we used a script to execute PIT for each test class as an input. We used the *targetTests* flag to specify this. We exported all results in *XML* and *HTML* formats and wrote a tool to aggregate the results from all *XML* outputs. Once the results were aggregated, we analysed them using R version 3.5.1. We make all scripts and the aggregation tool used in this study publicly available[3] for other researchers to replicate or expand our results.

4 RESULTS AND DISCUSSION

4.1 RQ1: The overall effectiveness of tests in the 10 open source systems

To get a better understanding of the results we first analyse the overall figures obtained from mutation testing. Table 3 shows the overall numbers for each dataset. *MutTests* is the total number of test classes associated with source code for which mutants (i.e. faults) were created. *TotalMut* is the total number of mutants created by PIT, whilst *Killed* shows the count of *TotalMut* that were detected. Finally, *mScore* is the ratio between the *Killed* and *TotalMut*. Figure 1 presents the distribution of killed and survived mutants in more detail.

The overall figures suggest that in the best case less than two thirds of all mutants were detected. *Webmagic* achieved the highest mutation score of 62% (i.e. about 3 in 5 faults were detected by the tests), however the same project has a relatively small number of generated mutants (only 253). Larger projects seem to achieve lower mutation scores which rarely go past 0.5. The exception to this is *activiti*, where nearly 30000 mutants were detected achieving the mutation score of 0.55. To check if there is any relationship between the quantity of tests and mutation score we use a ratio of the number of test cases per class (*TCperCl*). Table 4 shows this relationship.

Table 3: Mutation information for each dataset

Project	Version	MutTests	TotalMut	Killed	mScore
junit4	4.12	297	32501	9719	0.30
dropwizard	1.3.8	58	240	118	0.49
guice	4.2.2	633	81182	15985	0.20
metrics	4.0.5	74	595	232	0.39
jsoup	1.11.3	441	38871	5564	0.14
zxing	3.3.3	164	5448	2985	0.55
druid	1.11.1	1346	97151	27069	0.28
activiti	7.0.9	1426	53050	29103	0.55
retrofit	2.5.0	49	235	128	0.54
webmagic	0.7.3	20	253	156	0.62

Table 4: Test cases per class information.

Project	Classes	TestCases	TCperCl	mScore	Coverage
junit4	229	1463	6.39	0.30	86.80
jsoup	149	738	4.95	0.14	79.80
retrofit	216	706	3.27	0.54	48.50
druid	1225	1986	1.62	0.28	73.20
zxing	287	386	1.34	0.55	78.80
guice	635	848	1.34	0.20	81.90
activiti	1594	2097	1.32	0.55	66.80
metrics	206	226	1.10	0.39	80.80
dropwizard	537	320	0.60	0.49	86.40
webmagic	196	49	0.25	0.62	39.20

[3]https://figshare.com/s/6ed9a2cfa72db88b5976

The two projects with the highest number of test cases per class, i.e. *junit* and *jsoup*, achieved relatively low mutation scores. Similarly, *webmagic* is the project with the lowest number of test cases per class, but achieved the highest mutation score. These findings suggest that more tests does not necessarily mean more effective testing.

RQ1. What is the overall effectiveness of tests to detect faults in 10 open source systems? Overall, the open source systems in this study achieved relatively low mutation scores. The systems with the highest mutation scores are not necessarily those with the highest coverage or the highest number of test cases per class.

4.2 RQ2: The least and most detected fault types in the 10 open source test suites

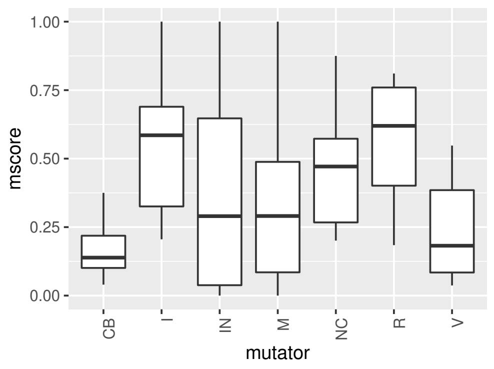

Figure 2: Overall mutation scores across the datasets per mutator

Figure 2 shows the average mutation scores for each of the seven fault types across the 10 datasets. Figure 2 suggests that some fault types are undetected more often than others (e.g. *CB* = conditional boundary and *V* = void method call). On the other hand, other fault types achieved median mutation scores of over 50% (e.g. *I* = increment and *R* = return value). To confirm whether these differences are significant, we conducted the Kruskal-Wallis non-parametric test for multiple groups. We obtained the *p*-value of $p = 0.006244$ with a confidence level of 95%. The *p*-value indicates that there is a significant difference in the mutation scores for different fault types. To further investigate the differences, we performed a post-hoc analysis where we employed the pairwise Wilcox test. As we compared seven different groups we used the Bonferroni correction to adjust the *p*-values. Table 5 depicts the *p*-values which show whether the difference between each pair of a fault type is significant or not.

The post-hoc pairwise comparison presented in Table 5 suggests that *CB* is the only fault type where a significant difference is observed. In particular, *CB* obtains significantly lower *mscore* compared to *NC* (negate conditionals) and *R* (return values). The void method call mutation operator is another fault type that achieves a low *mscore* (< 0.25), however due to a greater dispersion of values across the different datasets, its *mscore* is not significantly lower than those of the other fault types.

Table 5: Pairwise Wilcox test with Bonferroni correction

Mutator	CB	I	IN	M	NC	R
I	0.10					
IN	1.00	1.00				
M	1.00	1.00	1.00			
NC	0.02	1.00	1.00	1.00		
R	0.01	1.00	1.00	1.00	1.00	
V	1.00	0.29	1.00	1.00	0.60	0.06

Figure 3 presents the distribution of the fault types for each individual system. Apart from *junit4*, Figure 3 suggests that *CB* is indeed a fault type that often achieves the lowest *mscore* across the datasets. Figure 1 complements this result suggesting that the number of CB mutants created is generally aligned with the number of generated mutants for other fault types. An exception to this seems to be the *dropwizard*, *retrofit* and *webmagic* datasets. These datasets appear to have a disproportional number of killed *CB* mutants in comparison with *CB* in the other datasets. However, these three datasets have the lowest number of generated mutants and no mutants for the increment (*I*) and invert negatives (*IN*) fault types.

Finally, we wanted to check the magnitude of the detection rate (*mscore*) for each fault type across different datasets. To do that, we choose the average *mscore* calculated from the seven fault types as a baseline. The average *mscore* values are calculated for each individual dataset. We then plot the *mscore* values of each fault type against the average baseline. Figure 4 shows the relative distance of *mscore* for each fault type from the average *mscore* on a logarithmic scale. A logarithmic scale is best to represent ratio values [39]. Large deviations from the baseline indicate that a particular fault type is either over-detected (above the baseline) or under-detected (below the baseline) on average. From Figure 4 large relative differences can be observed for *CB*. For example, in the case of *retrofit*, *CB* achieves an *mscore* which is about 10 times (10^{-1}) below the baseline. One contributing factor to this could be the lower average value of *mscore* for this dataset, as no faults were generated for the types *I*, *IN* and *M*. However, it appears that the particularly low *mscore* in *retrofit* further extends to *dropwizard* and *webmagic*. In addition, the *mscore* of *CB* is below average for all the datasets. These results suggest that improving boundary checks in tests could improve test effectiveness.

In addition to *CB*, there are several projects where the kill-rate for the void method call mutator (*V*) was substantially lower than other fault types. Most notably, *V* is particularly low in *junit4* and *guice*. These projects would benefit from tests with improved checking of side effects. In contrast, it is interesting to observe that some fault types were constantly detected more often than others. For example, the return value mutator (*R*) was generally detected by tests. This is not too surprising as at a minimum the method results should tested. Despite the high detection rates for *R*, it would still be worth investigating why not all *R* mutators were killed as that

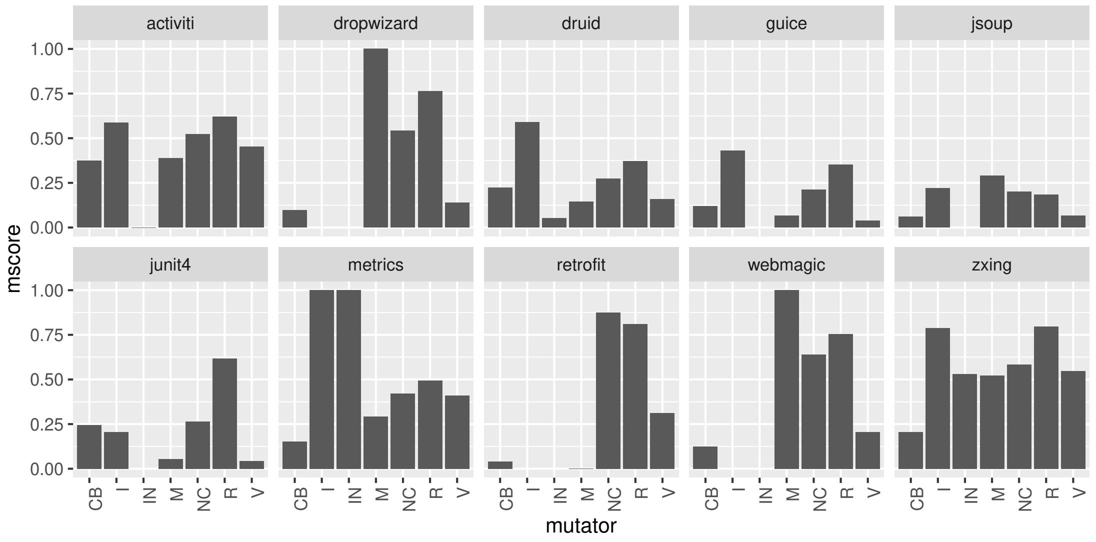

Figure 3: Mutation scores for the different fault types across all datasets. Where the bars are missing no faults of that type were generated.

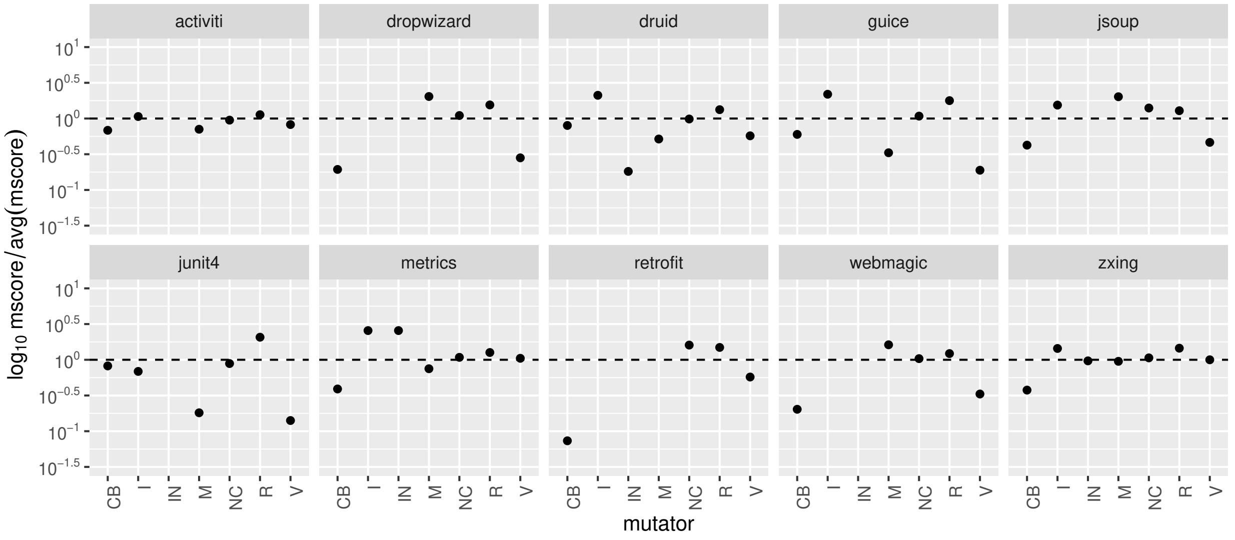

Figure 4: Relative distances of the individual fault type mutation scores from the average scores by dataset. Where the dots are missing no faults of that type were generated.

could further improve tests. Other mutators, such as maths and negate conditional were generally well detected by the tests.

RQ2. Which of seven fault types are least and most often detected by tests in these 10 open source systems? The conditional boundary fault type seems most likely to slip through testing followed by the void mutator. The return, maths, negate conditional and increment mutators are typically more often detected by tests.

4.3 RQ3: Improving test effectiveness by exploiting a non uniform distribution of different fault types

Compared to other fault types, Table 5 indicates that *CB* is the only amongst the seven for which exist a significant difference in *mscore* (i.e. *CB* is significantly missed by tests compared to *NC* and *R*, $p < 0.05$). In the case of CB, tests seem to often miss differences in the conditional boundaries of the system under test. The snippets of Code 1 and 2 demonstrate one example of that. Where Code 1 is the original version of functional code, Code 2 represents a slight variation where the boundary condition is changed. Our results suggests that tests often fail to detect this type of mistake. This is surprising since boundary problems have been known about for many years.

Insufficient testing of boundary conditions can have consequences that span from minor issues to more serious security threats. For example, iterating through an array of different possible choices can lead to an item of the array at the boundary not being checked. This type of mistake can often be quickly observed and fixed. However, in other cases, insufficient boundary checks can lead to major security issues such as buffer overflow. Alhazmi et al. showed that a significant number of high security vulnerabilities are caused by buffer overflows which are linked to boundary condition problems [3]. Therefore, it is of utmost importance that boundary conditions are sufficiently tested. Our results suggest that this can largely be improved by adding additional test cases that check boundary conditions.

Our results suggest that another type of fault commonly missed by tests is caused by the removal of void method calls. According to our results, more than 75% of tests exercising a system under test containing void method calls are affected by this potential issue. We suspect that some faults which are not caught by tests could relate to side effects caused by removing void method calls. Side effects happen when a method modifies a non-local variable, which can have an effect on other parts of code that use the same non-local variable. As shown in Code 3, side effects can have consequences that go beyond the system under test. In this example, the `calculate()` method represents the system under test. Code 4 depicts an example of a test that checks its functionality. The removal of `closeAccount()` from `calculate()` in Code 3 will not have any effect on the unit test in Code 4. However, the effect on the rest of the system can be significant. Any other part of code which relies on the variable `closed` is now affected. Therefore, unit tests may not be an appropriate way to comprehensively test the `calculate()` method. Instead, more sophisticated integration tests should be employed.

RQ3. How can tests be improved to ensure the seven fault types covered in this study are effectively detected? We show two common types of faults that are likely to slip through testing and suggest practical improvements to tests. In particular, unit tests should more comprehensively check boundary cases, whilst integration tests should be used in situations where side effects can have impact beyond the system under test.

5 RELATED WORK

One of the main goals in software testing research is to establish ways in which tests can be effective, i.e. capable of exposing faults. Traditionally, code coverage has been used as a proxy for measuring test effectiveness, making it a defacto indicator of testing effectiveness. The rationale being that tests which cover more lines and independent paths in the code should be more effective in exposing faults. However, an increasing number of studies have shown that coverage is a weak proxy for test effectiveness. For example, Antinyan et al. [6] demonstrated that the increase in code coverage produced only a slight tendency for decreasing the number of faults in a large telecommunication system (>2M LOC). Notably, in their study the size of modules and their change rate were controlled for. Kochhar et al. [22] arrived at a similar conclusion, i.e. there is no significant correlation between coverage and post-release faults. It is likely that the factors which impact test effectiveness are complex and involves multiple factors.

The ability of tests to find specific types of fault has been studied previously. Deng et al. empirically investigated the fault detection rate of the Statement Deletion operator [14]. They reported a reduction of 80% in the number of generated mutants with only a modest loss in test effectiveness. Delemaro et al. studied the Interface Mutation operators specifically designed for integration testing to assess their test effectiveness [12]. Their approach captured nearly all seeded faults caused by an incorrect interaction between modules.

Code 1: Original code

```
if (a <= b) {
   //some code
}
```

Code 2: Faulty code

```
if (a < b) {
   //some code
}
```

Code 3: Code with side effects

```
boolean closed = false;
double calculate(double price){
  closeAccount();
  return price*1.2;
}
void closeAccount(){
  closed = true;
}
```

Code 4: Accompanying unit test for the calculate method

```
@Test
void testCalculate(){
  double expected = 12.0;
  double actual = calculate(10.0);
  assertEquals(expected, actual, 0.01);
}
```

Smith and Williams empirically evaluated multiple mutation operators using a small back-end web application [36]. They found that the usefulness of operators depend on the context, as for their web-based application conditional operators were particularly useful compared to arithmetic operators which were seldom used in the system. In this study we also found that the conditional-based faults need more testing. Petrović and Ivanković used mutation testing in Google to analyse which fault types get detected most often in their commercial systems [32]. They report that the survival rate amongst different fault types is stable and does not vary significantly. In addition, Schwartz et al. investigated the relationship between different fault types and test effectiveness to find that some faults are detected more often than others [35].

Other factors that have been reported in the literature to influence test effectiveness are test suite size, the number of covered methods per test and the number of assertions. Inozemstva and Holmes studied the effect of test suite size on test effectiveness and discovered that coverage has moderate to high correlation if the size of a test suite is not controlled [19]. Petrić et al. established that robust tests (i.e. tests that cover multiple methods) are more likely to expose real faults in the system [31]. Zhang and Mesbah on the other hand demonstrated that the number of assertions in tests is a strong indicator of test effectiveness [42], which aligns with the previous finding that the robustness of tests makes them more effective. In addition, Bowes et al. identified several other factors, such as test maintainability and comprehensibility, influence test effectiveness [10]. Our results align with these previous findings. In particular, Bowes et al. suggested that developers need to write more happy and sad tests (i.e. tests that verify and that break the system) [10], as such tests increase the likelihood of covering border cases.

In this study we investigated fault type as the potential factor that influences test effectiveness. Fault type provides a finer grained understanding of the shortcomings in the test suite by pinpointing weaknesses in the tests. For instance, when a particular fault type is more prominent in the system it is possible to improve testing to be more effective in capturing that specific fault type. The work of Schwartz et al. considered the relationship between fault types and test effectiveness for systems satisfying high coverage [35]. Using the MuJava mutation tool, they found that the arithmetic and relational operators replacement are more often detected compared to other 17 traditional mutation operators they investigated. Contrary, the conditional operator used in this study, which is most similar to the MuJava's relational operator replacement, is the least often detected in the 10 open source systems. However, in our work we focus our selection of datasets and tools to explicitly address the equivalence problem without making presumptions on how well code is covered. As mentioned in the background section, reducing the equivalence problem is important to fairly compare different fault types and their chance of being exposed by tests.

Our study sheds light on the additional factors that can improve the detection of faults by tests. Most prominently, boundary checks are undertested. Our results suggest that tests which robustly check boundaries would improve test effectiveness. Similarly, covering cases caused by side effects in method calls would improve the overall effectiveness of the test suite.

6 THREATS TO VALIDITY

Here we consider the potential internal and external threats to validity and our approaches to mitigate them. We used mutants as surrogates for real faults in this study. Even though mutants might be different from real faults, we believe that potential issues that mutants reveal are worth considering in order to improve testing.

Another internal threat to this type of study is caused by equivalent mutants. As explained in the background and methodology, equivalent mutants can deflate mutation score values [28]. Yao et al. demonstrated that some mutation operators are considerably more prone to equivalent mutants than others [40]. Without considering their impact, equivalent mutants could significantly effect the results of this type of study. We employed techniques to reduce the number of equivalent mutants. First, amongst the available mutation testing tools for Java, we selected PIT which has been shown to generate the least number of equivalent mutants [21]. Second, we restricted ourselves to the seven mutants that are most unlikely to generate equivalent mutants[4].

The selection of the datasets in our study might pose threats to external validity. To minimise this threat, we carefully selected candidates to be included in our study. We ensured that only active systems which are widely accepted and used by the community are considered. In addition, we selected a diverse set of systems to check whether our findings are local to the project or can be generalised to a variety of different systems. We believe the selected systems provide suitable diversity and are worthy of investigation.

7 CONCLUSIONS

By investigating the detection rate of different fault types, we show that some types of faults are more likely to slip through testing than others. In particular, conditional boundaries and faults relating to potential side effects are more likely to be undetected by tests in the 10 systems we investigated. Relying only on traditional test effectiveness metrics is not sufficient to reveal those issues which often go undetected. Our results suggest several practices to improve test effectiveness. Developers should pay more attention to boundary cases when writing tests. One approach is to write tests that both, verify and break the system, to increase the coverage of border cases. Developers should also look out for potential side effects in the system under test. Developers need to understand the consequences of particular code execution on the system. Our results indicate that developers also need to consider trivial cases, such as attending to a function's return value. Even though the return mutator is an obvious fault and it is detected frequently, there are a considerable number of cases where this type of fault goes undetected.

Researchers may also benefit from this research. More work is needed to experiment with techniques for reducing the equivalent mutant problem in order to include a greater number of fault types. It is likely that by including more fault types we can get a broader understanding of other common factors that can improve test effectiveness. Further investigation of factors where tests are more successful in finding a particular fault type could also increase the understanding of what makes some tests more successful than others in finding faults. Finally, we believe it is worth investigating tests where faults related to the return values go undetected. Better understanding of those tests would likely be beneficial for improving test effectiveness.

REFERENCES

[1] [n.d.]. Mutation operators. http://pitest.org/quickstart/mutators/. (Accessed on 02/20/2019).

[2] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. 2016. Can Testedness Be Effectively Measured?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* 547–558. https://doi.org/10.1145/2950290.2950324

[3] Omar H Alhazmi, Sung-Whan Woo, and Yashwant K Malaiya. 2006. Security vulnerability categories in major software systems.. In *Communication, Network, and Information Security.* 138–143.

[4] http://pitest.org/quickstart/basic_concepts/

[4] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978 – 2001. https://doi.org/10.1016/j.jss.2013.02.061

[5] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments? [software testing]. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 402–411. https://doi.org/10.1109/ICSE.2005.1553583

[6] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. 2018. Mythical Unit Test Coverage. *IEEE Software* 35, 3 (May 2018), 73–79. https://doi.org/10.1109/MS.2017.3281318

[7] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (Nov 2014), 1100–1125. https://doi.org/10.1109/TSE.2014.2342227

[8] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (01 Aug 2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[9] Hudson Silva Borges and Marco Tulio Valente. 2017. Application Domain of 5,000 GitHub Repositories. https://doi.org/10.5281/zenodo.804474

[10] David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan. 2017. How Good Are My Tests?. In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics* (Buenos Aires, Argentina) *(WETSoM '17)*. IEEE Press, Piscataway, NJ, USA, 9–14. https://doi.org/10.1109/WETSoM.2017..2

[11] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[12] M. E. Delamaro, J. C. Maidonado, and A. P. Mathur. 2001. Interface Mutation: an approach for integration testing. *IEEE Transactions on Software Engineering* 27, 3 (March 2001), 228–247. https://doi.org/10.1109/32.910859

[13] D. Delgado and A. Martinez. 2013. Cost Effectiveness of Unit Testing: A Case Study in a Financial Institution. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* 340–347. https://doi.org/10.1109/ESEM.2013.50

[14] L. Deng, J. Offutt, and N. Li. 2013. Empirical Evaluation of the Statement Deletion Mutation Operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* 84–93. https://doi.org/10.1109/ICST.2013.20

[15] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia. 2017. Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (June 2017), 202–212. https://doi.org/10.1109/TETCI.2017.2699224

[16] R. Gopinath, C. Jensen, and A. Groce. 2014. Mutations: How Close are they to Real Faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering.* 189–200. https://doi.org/10.1109/ISSRE.2014.40

[17] R. Gopinath, B. Mathis, and A. Zeller. 2018. If You Can't Kill a Supermutant, You Have a Problem. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 18–24. https://doi.org/10.1109/ICSTW.2018.00023

[18] Atul Gupta and Pankaj Jalote. 2008. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer* 10, 2 (01 Mar 2008), 145–160. https://doi.org/10.1007/s10009-007-0059-5

[19] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[20] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[21] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (01 Aug 2018), 2426–2463. https://doi.org/10.1007/s10664-017-9582-5

[22] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan. 2017. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* 66, 4 (Dec 2017), 1213–1228. https://doi.org/10.1109/TR.2017.2727062

[23] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 142–151. https://doi.org/10.1109/ICSTW.2016.41

[24] W. B. Langdon, M. Harman, and Y. Jia. 2009. Multi Objective Higher Order Mutation Testing with Genetic Programming. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques.* 21–29. https://doi.org/10.1109/TAICPART.2009.18

[25] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *3rd International Symposium on Empirical Software Engineering and Measurement.* 291–301. https://doi.org/10.1109/ESEM.2009.5315981

[26] Akbar Siami Namin and Sahitya Kakarla. 2011. The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. 342–352. https://doi.org/10.1145/2001420.2001461

[27] A. Panichella, R. Oliveto, and A. De Lucia. 2014. Cross-project defect prediction models: L'Union fait la force. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE).* 164–173. https://doi.org/10.1109/CSMR-WCRE.2014.6747166

[28] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 354–365. https://doi.org/10.1145/2931037.2931040

[29] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 936–946. http://dl.acm.org/citation.cfm?id=2818754.2818867

[30] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, Atif M. Memon (Ed.). Advances in Computers, Vol. 112. Elsevier, 275 – 378. https://doi.org/10.1016/bs.adcom.2018.03.015

[31] Jean Petrić, Tracy Hall, and David Bowes. 2018. How Effectively Is Defective Code Actually Tested?: An Analysis of JUnit Tests in Seven Open Source Systems. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering* (Oulu, Finland) *(PROMISE'18)*. ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/3273934.3273939

[32] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. ACM, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521

[33] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 47–53. https://doi.org/10.1109/ICSTW.2018.00027

[34] R. Purushothaman and D. E. Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31, 6 (June 2005), 511–526. https://doi.org/10.1109/TSE.2005.74

[35] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. 2018. Investigating faults missed by test suites achieving high code coverage. *Journal of Systems and Software* 144 (2018), 106 – 120. https://doi.org/10.1016/j.jss.2018.06.024

[36] B. H. Smith and L. Williams. 2007. An Empirical Evaluation of the MuJava Mutation Operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007).* 193–202. https://doi.org/10.1109/TAIC.PART.2007.12

[37] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[38] M. Waterloo, S. Person, and S. Elbaum. 2015. Test Analysis: Searching for Faults in Tests (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 149–154. https://doi.org/10.1109/ASE.2015.37

[39] Claus O. Wilke. 2018. *Fundamentals of Data Visualization: A primer on making informative and compelling figures.* O'Reilly Media.

[40] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 919–930. https://doi.org/10.1145/2568225.2568265

[41] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software test-code engineering: A systematic mapping. *Information and Software Technology* 58 (2015), 123 – 147. https://doi.org/10.1016/j.infsof.2014.06.009

[42] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 214–224. https://doi.org/10.1145/2786805.2786858