An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption

Thierry Titcheu Chekam*, Mike Papadakis*, Yves Le Traon*, and Mark Harman†
*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
†University College London and Facebook, London, UK
Email:thierry.titcheu-chekam@uni.lu, michail.papadakis@uni.lu, yves.letraon@uni.lu and mark.harman@ucl.ac.uk

Abstract—Many studies suggest using coverage concepts, such as branch coverage, as the starting point of testing, while others as the most prominent test quality indicator. Yet the relationship between coverage and fault-revelation remains unknown, yielding uncertainty and controversy. Most previous studies rely on the Clean Program Assumption, that a test suite will obtain similar coverage for both faulty and fixed ('clean') program versions. This assumption may appear intuitive, especially for bugs that denote small semantic deviations. However, we present evidence that the Clean Program Assumption does not always hold, thereby raising a critical threat to the validity of previous results. We then conducted a study using a robust experimental methodology that avoids this threat to validity, from which our primary finding is that strong mutation testing has the highest fault revelation of four widely-used criteria. Our findings also revealed that fault revelation starts to increase significantly only once relatively high levels of coverage are attained.

Index Terms—Mutation testing, test effectiveness, code coverage, real faults, test adequacy

I. INTRODUCTION

The question of which coverage criterion best guides software testing towards fault revelation remains controversial and open [1]–[5]. Previous research has investigated the correlation between various forms of structural coverage and fault revelation using both real and simulated faults (seeded into the program under test as mutants). Determining the answer to the test coverage question is important because many software testing approaches are guided by coverage [6]–[9], and the industry standards used by practising software engineers mandate the achievement of coverage [10], [11]. Nevertheless, the findings of the studies hitherto reported in the literature have been inconclusive, with the overall result that this important question remains unanswered.

Most previous studies make an important assumption, the veracity of which has not been previously investigated. We call this assumption the 'Clean Program Assumption'. The assumption is that test suites are assessed based on the coverage they achieve on clean programs, which do not contain any known faults. This practice might be problematic when using faulty versions (in order to check the fault-revealing potential of the test suites) since test suites are assessed on each of the faulty versions and not the clean program from which (and for which) the coverage was measured.

Of course, it is comparatively inexpensive (and therefore attractive to experimenters) to use a single test suite for the clean program, rather than using separate test suites for each of the faulty versions. However, a test suite that is adequate for the clean program may be inadequate for some of the faulty versions, while test suites that have been rejected as inadequate for the clean program may turn out to be adequate for some of the faulty versions. Furthermore, the coverage achieved by inadequate test suites may differ between the clean version of the program and each of its faulty versions.

These differences have not previously been investigated and reported upon; if they prove to be significant, then that would raise a potential threat to the scientific validity of previous findings that assume the Clean Program Assumption. We investigated this assumption and found strong empirical evidence that, it does *not* always hold; there are statistically significant differences between the coverage measurements for clean and faulty versions of the programs we studied.

Given that we found that we cannot rely on the Clean Program Assumption, we then implemented a robust methodology, in which the test suite for each test adequacy criteria is recomputed for each of the faulty versions of the program under test. We studied statement, branch, strong and weak mutation criteria, using a set of real-world faults, recently made available [12], located in 145,000 lines of C code spread over four different real-world systems.

We used systems with mature test suites, which are augmented, both by using the popular test data generation tool KLEE [13] and by hand, to ensure the availability of a high quality pool of test data from which to draw test suites. Unfortunately, such a high quality test pool cannot yet be guaranteed using automated test data generation tools alone, partly because of the inherent undecidability of the problem, and partly because of the limitations of current tools [14], [15]. Nevertheless, it is important for us to have such a high quality test pool in order to allow us to sample multiple test suites (related to the faults studied) and to achieve different experimentally-determined levels of coverage, when controlling for test suite size. Using randomised sampling from the augmented test pool, we were able to generate test suites that achieve many different coverage levels, thereby placing coverage level under experimental control.

Perhaps the most surprising result from our study is that we find evidence for a strong connection only between coverage attainment and fault revelation for one of the four coverage criteria: strong mutation testing. For statement, branch and weak mutation testing, we found that increased coverage has little or no effect on fault revelation. This is a potentially important finding, notwithstanding the 'Threats to Validity' of generalisation discussed at the end of this paper, especially given the emphasis placed on branch coverage by software tools and industrial standards. While some previous studies have made similar claims (for branch and block coverage [16]), these conclusions were subsequently contradicted [1], [4], [17], [18].

One of the other interesting (and perhaps surprising) findings of our study is that the relationship between strong mutation and fault revelation exhibits a form of 'threshold behaviour'. That is, above a certain threshold, we observed a strong connection between increased coverage and increased fault revelation. However, below this threshold level, the coverage achieved by a test suite is simply irrelevant to its fault-revealing potential. This 'threshold observation' and the apparent lack of connection between fault revelation and statement/branch/weak mutation coverage may go some way to explaining some of the dissimilar findings from previous studies (and may partially reduce the apparent controversy). According to our results, any attempt to compare inadequate test suites that fail to reach threshold coverage may be vulnerable to 'noise effects': two studies with below-threshold coverage may yield different findings, even when the experimenters follow identical experimental procedures.

More research is needed in this important area to fully understand this fundamental aspect of software testing, and we certainly do not claim to have completely answered all questions in this paper. We do, however, believe our findings significantly improve our understanding of coverage criteria, their relationship to each other and to fault revelation. Our primary contributions are to expose and refute the Clean Program Assumption, and to present the results of a larger-scale empirical study that does not rest on this assumption. The most important finding from this more robust empirical study is the evidence for the apparent superiority of strong mutation testing and the observation of threshold behaviour, below which improved coverage has little effect on fault revelation.

II. TEST ADEQUACY CRITERIA

Test adequacy criteria define the requirements of the testing process [19]. Goodenough and Gerhart [20] define test criteria as predicates stating that a criteria captures "what properties of a program must be exercised to constitute a thorough test, i.e., one whose successful execution implies no errors in a tested program". As a result, they guide testers in three distinct ways [19]: by pointing out the elements that should be exercised when designing tests, by providing criteria for terminating testing (when coverage is attained), and by quantifying test suite thoroughness.

Although there is a large body of work that crucially relies upon test criteria [15], [21], there remain comparatively few studies in the literature that address questions related to actual fault revelation (using real faults) to reliably confirm the coverage-based assessment of test thoroughness. We therefore, empirically examine the ability of criteria-guided testing in uncovering faults. We investigate four popular test criteria: the main two structural criteria (namely statement and branch testing), and the main two fault-based criteria (namely weak and strong mutation testing).

A. Statement and Branch Adequacy Criteria

Statement testing (aka statement coverage) relies on the idea that we cannot be confident in our testing if we do not, at least, exercise (execute) every reachable program statement at least once. This practice is intuitive and is widely-regarded as a (very) minimal requirement for testing. However, programs contain many different types of elements, such as predicates, so faults may be exposed only under specific conditions, that leave them undetected by statement adequate test suites. Therefore, stronger forms of coverage have been defined [19]. One such widely-used criteria, commonly mandated in industrial testing standards [10], [11] is branch coverage (or branch testing). Branch testing asks for a test suite that exercises every reachable branch of the Control Flow Graph of the program. Branch testing is stronger to statement testing, which only asks for a test suite that exercises every node of the graph.

B. Mutation-Based Adequacy Criteria

Mutation testing deliberately introduces artificially-generated defects, which are called 'mutants'. A test case that distinguishes the behaviour of the original program and its mutant is said to 'kill' the mutant. A mutant is said to be weakly killed [9], [31], [32], if the state of computation immediately after the execution of the mutant differs from the corresponding state in the original program. A mutant is strongly killed [9], [31], [32] if the original program and the mutant exhibit some observable difference in their output behaviour. Strong mutation does not subsume weak mutation because of potential failed error propagation [9], [33], which may cause state differences to be over-written by subsequent computation.

For a given set of mutants, M, mutation coverage entails finding a test suite that kills all mutants in M. The proportion of mutants in M killed by a test suite T is called the mutation score of T. It denotes the degree of achievement of mutation coverage by T, in the same way that the proportion of branches or statements covered by T denotes its degree of branch or statement adequacy respectively.

Previous research has demonstrated that mutation testing results in strong test suites, which have been empirically observed to subsume other test adequacy criteria [32]. There is also empirical evidence that mutation score correlates with actual failure rates [4], [30] indicating that, if suitable experimental care is taken, then these artificially-seeded faults can be used to assess the fault revealing-potential of test suites.

TABLE I

Summary of previous studies on the relationship of test criteria and faults.

Author(s) [Reference]	Year	Largest Subject	Language	Test Criterion	Fault Types	Summary of Primary Scientific Findings
Frankl & Weiss [22], [23]	'91, '93	78	Pascal	branch, all-uses	real faults	All-uses relates with test effectiveness, while branch does not.
Offutt et al. [24]	'96	29	Fortran, C	all-uses, mutation	seeded faults	Both all-uses and mutation are effective but mutation reveals more faults.
Frankl et al. [25]	'97	78	Fortran, Pascal	all-uses, mutation	real faults	Test effectiveness (for both all-uses and mutation) is increasing at higher coverage levels. Mutation performs better.
Frankl & Iakounenko [5]	'98	5,000	C	all-uses, branch	real faults	Test effectiveness increases rapidly at higher levels of coverage (for both all-uses and branch). Both criteria have similar test effectiveness.
Briand & Pfahl [16]	'00	4,000	C	block, c-uses, p-uses, branch	simulation	There is no relation (independent of test suite size) between any of the four criteria and effectiveness
Andrews et al. [4]	'06	5,000	C	block, c-uses, p-uses, branch	real faults	Block, c-uses, p-uses and branch coverage criteria correlate with test effectiveness.
Namin & Andrews [17]	'09	5,680	C	block, c-uses, p-uses, branch	seeded faults	Both test suite size and coverage influence (independently) the test effectiveness
Li et al. [26]	'09	618	Java	prime path, branch, all-uses, mutation	seeded faults	Mutation testing finds more faults than prime path, branch and all-uses.
Papadakis & Malevris [9]	'10	5,000	C	Mutant sampling, 1st & 2nd order mutation	seeded faults	1st order mutation is more effective than 2nd order and mutant sampling. There are significantly less equivalent 2nd order mutants than 1st order ones.
Ciupa et al. [27]	'09	2,600	Eiffel	Random testing	real faults	Random testing is effective and has predictable performance.
Wei et al. [28]	'12	2,603	Eiffel	Branch	real faults	Branch coverage has a weak correlates with test effectiveness.
Hassan & Andrews [29]	'13	16,800	C, C++, Java	multi-Point Stride, data flow, branch	mutants	Def-uses is (strongly) correlated with test effectiveness and has almost the same prediction power as branch coverage. Multi-Point Stride provides better prediction of effectiveness than branch coverage.
Gligoric et al. [1], [18]	'13, '15	72,490	Java, C	AIMP, DBB, branch, IMP, PCC, statement	mutants	There is a correlation between coverage and test effectiveness. Branch coverage is the best measure for predicting the quality of test suites.
Inozemtseva & Holmes [3]	'14	724,089	Java	statement, branch, modified condition	mutants	There is a correlation between coverage and test effectiveness when ignoring the influence of test suite size. This is low when test size is controlled.
Just et al. [30]	'14	96,000	Java	statement, mutation	real faults	Both mutation and statement coverage correlate with fault detection, with mutants having higher correlation.
Gopinath et al. [2]	'14	1,000,000	Java	statement, branch, block, path	mutants	There is a correlation between coverage and test effectiveness. Statement coverage predicts best the quality of test suites.
This paper	'17	83,100	C	statement, branch, weak & strong mutation	real faults	There is a strong connection between coverage attainment and fault revelation for strong mutation but weak for statement, branch and weak mutation. Fault revelation improves significantly at higher coverage levels.

C. Previous Empirical Studies

Table I summarises the characteristics and primary scientific findings of previous studies on the relationship between test criteria and fault detection. As can be seen, there are three types of studies, those that use real faults, seeded faults and mutants. Mutants refer to machine-generated faults, typically introduced using syntactic transformations, while seeded faults refer to faults placed by humans.

One important concern regards the Clean Program Assumption when using either seeded or mutant faults or both. In principle most of the previous studies that used seeded or mutant faults assume the Clean Program Assumption as their experiments were performed on the original (clean) version and not on the faulty versions. This is based on the intuitive assumption that as artificial faults denote small syntactic changes they introduce small semantic deviations. Our work shows that this assumption does not hold in the case of real faults and thus, leaves the case of artificial faults open for future research. Though, previous research has shown that higher order (complex) mutants [9], [34] are generally weaker than first order (simple) ones and that they exhibit distinct behaviours [35], which implies that the assumption plays an important role in the case of artificial faults.

Only the studies of Frankl and Weiss [22], [23], Frankl et al. [25], Frankl and Iakounenko [5], Ciupa et al. [27] and Wei et al. [28] do not assume the Clean Program Assumption. Unfortunately, all these studies have limited size and scope of their empirical analysis and only the work of Frankl et al. [25] investigates mutation. Generally, only three studies (Offutt et al. [24], Frankl et al. [25] and Li et al. [26]) investigate the fault revelation question for mutation, but all of them use relatively small programs and only the work of Frankl et al. [25], uses real faults, leaving open the questions about the generalisability of their findings.

The studies of Andrews et al. [4] and Just et al. [30] used real faults to investigate whether mutants or other criteria can form substitutes for faults when conducting test experiments. This question differs from the fault revelation question because it does not provide any answers concerning test criteria fault revelation. Also, both these studies make the Clean Program Assumption and do not control for test suite size.

Overall, although the literature contains results covering a considerable number of test adequacy criteria, including the most popular (branch, statement and mutation-based criteria), our current understanding of these relationships is limited and rests critically upon the Clean Program Assumption.

III. RESEARCH QUESTIONS

Our fist aim is to investigate the validity of the 'Clean Program Assumption', since much of our understanding of the relationships between test adequacy criteria rests upon the validity of this assumption. Therefore, a natural first question to ask is the extent to which experiments with faults, when performed on the "clean" (fixed) program versions, provide results that are representative of those that would have been observed if the experiments had been performed on the "faulty" program versions. Hence we ask:

RQ1: *Does the 'Clean Program Assumption' hold?*

Given that we did, indeed, find evidence to reject the Clean Program Assumption, we go on to investigate the relationship between achievement of coverage and fault revelation, using a more robust experimental methodology that does not rely upon this assumption. Therefore, we investigate:

RQ2: *How does the level of fault revelation vary as the degree of the coverage attained increases?*

Finally, having rejected the Clean Program Assumption, and investigated the relationship between fault revelation for adequate and partially adequate coverage criteria, we are in a position to compare the different coverage criteria to each other. Therefore we conclude by asking:

RQ3: *How do the four coverage criteria compare to each other, in terms of fault revelation, at varying levels of coverage?*

The answers to these questions will place our overall understanding of the fault-revealing potential of these four widely-used coverage criteria on a firmer scientific footing, because they use real-world faults and do not rely on the Clean Program Assumption.

IV. RESEARCH PROTOCOL

Our study involves experiments on mature real-world projects, with complex real faults, developer, machine-generated and manually-written tests. All these tests yields a pool from which we sample, to experimentally select different coverage levels, while controlling for test suite size (number of test cases). Our experimental procedure follows the following five steps:

1) We used CoREBench, a set of real faults that have been manually identified and isolated, using version control and bug tracking systems in the previous work by Böhme and Roychoudhury [12]. Böhme and Roychoudhury with the introduction of CoREBench have created a publicly available set of real-world bugs on which others, like ourselves, can experiment.
2) We extracted the developer tests for each of the faults in CoREBench.
3) We generated test cases covering (at least partially) all the faults using the state-of-the-art dynamic symbolic execution test generation tool, KLEE [13], [36].
4) We manually augmented the developer and automatically generated test suites that were obtained in the previous steps. To do so we used the bug reports of the

TABLE II
THE SUBJECT PROGRAMS USED IN THE EXPERIMENTS. FOR EACH OF THEM, THE NUMBER OF TEST CASES (TC), THEIR SIZE IN LINES OF CODE AND NUMBER OF CONSIDERED FAULTS ARE PRESENTED.

Program	Size	Developer TC	KLEE TC	Manual TC	Faults
Coreutils	83,100	4,772	13,920	27	22
Findutils	18,000	1,054	3,870	7	15
Grep	9,400	1,582	4,280	37	15
Make	35,300	528	138	25	18

faults and generated additional test cases to ensure that each fault can potentially be revealed by multiple test cases from the test pool. The combined effect of Steps 2, 3 and 4 is to yield an overall test pool that achieves both high quality and diversity, thereby facilitating the subsequent selection step.
5) We perform statement, branch, weak and strong mutation testing, using multiple subsets selected from the test pool (constructed in the Steps 2, 3 and 4), using sampling with uniform probability. Test suites for varying degrees of coverage according to each one of the four criteria were constructed for all faulty programs, one per fault in CoREBench, in order to avoid the Clean Program Assumption.

A. Programs Used

To conduct our experiments it is important to use real-world programs that are accompanied by relatively good and mature test suites. Thus, we selected the programs composing the CoREBench [12] benchmark: "Make", "Grep", "Findutils", and "Coreutils". Their standardized program interfaces were helpful in our augmentation of the developers' initial test suites, using automated test data generation. Furthermore, the available bug reports for these programs were helpful to us in the laborious manual task of generating additional test cases.

Table II records details regarding our test subjects. The size of these programs range from 9 KLoC to 83KLoC and all are accompanied by developer test suites composed of numerous test cases (ranging from 528 to 4,772 test cases). All of the subjects are GNU programs, included in GNU operating systems and typically invoked from the command line (through piped commands). Grep is a tool that processes regular expressions, which are used for text matching and searching. The Make program automates the source code building process. Findutils and Coreutils are each collections of utilities for, respectively, searching file directories and manipulating files and text for the UNIX shell.

B. CoREBench: realistic, complex faults

To conduct this study we need a benchmark with real-world complex faults that can be reliably used to evaluate and compare the four coverage criteria we wish to study. Unfortunately benchmarks with real errors are scarce. CoREBench [12] is a collection of 70 systematically isolated faults, carefully extracted from the source code repositories and bug reports of the projects we study.

The most commonly-used benchmarks are the Siemens Suite and the Software Infrastructure Repository SIR [37], [38], but sadly neither can help us to answer our particular chosen research questions. While the Siemens suite has been widely used in previous studies, the degree to which generalisation is possible remains limited, because the programs are small, and cannot truly be said to be representative of real-world systems. The SIR repository overcomes this limitation, because it contains real-world programs, and is a very valuable resource. Nevertheless, many of the faults collected for the SIR programs are artificially seeded faults. This repository is thus less relevant to our study, because we seek to study the relationship between such artificially seeded faults and real faults as part of our set of research questions.

The CoREBench benchmark we chose to use was built by analysing 4,000 commits, which led to the isolation and validation (through test cases) of 70 faults [12]. Every fault was identified by exercising the project commits with validating test cases that reveal the faults. Thus, the test cases pass on the versions before the bug-introducing commit and fail after the commit. Also, the test cases pass again after the fixing commit. Further details regarding the benchmark can be found in the CoREBench paper by Böhme and Roychoudhury [12] and also on its accompanying website[1].

When conducting our analyses, we also verified the faulty and fixed versions using both the developer and additionally generated (either manually or automatically) test cases (details regarding the test suites we used can be found in Section IV-C). As the "faulty" and "fixed" program versions were mined from project repositories by analysing commits, they could have differences that are irrelevant to the faults we study. Thus, they could potentially bias our results because they might arbitrarily elevate the number of program elements to be covered (due to altered code unrelated to the fault). To avoid this, we checked and removed irrelevant code from the few cases we found, using the test suites as behaviour-preserving indicators (we used delta debugging [39] to minimise the differences between the "faulty" and "fixed" versions).

Finally, we excluded nine faults from our analysis due to technical problems. Faults with CoREBench identifiers 57 and 58 for the Make program failed to compile in our environment. Also we had technical problems forming the annotations for (Make) faults with identifiers 64 and 65 and thus, KLEE could not create additional test suites for these faults. Fault 42 of Grep, 33 and 37 of the Findutils and 60, 62 of Make took us so much execution time that we were forced to terminate their execution after 15 days.

C. Test Suites Used

The developer test suites for all the projects we studied were composed of approximately 58,131 tests in total. As these were not always able to find the faults (because in this case bugs would have been noticed before being reported), the authors of CoREBench designed test cases that reveal them (typically

only one test to expose each bug). However, we not only need to expose the bugs, but also to expose them multiple times in multiple different ways in order to allow our uniform test suite selection phase to benefit from a larger and more diverse pool from which to select.

Therefore, to further strengthen the test suites used in our study, we augment them in a two-phase procedure. In the first phase we used KLEE, with a relatively robust timeout limit of 600 seconds per test case, to perform a form of differential testing [40] called shadow symbolic execution [36], which generates 22,208 test cases. Shadow symbolic execution generates tests that exercise the behavioural differences between two different versions of a program, in our case the faulty and the fixed program versions. We guided shadow symbolic execution by manual annotations to the subject programs that have no side-effects.

Unfortunately, the current publicly available version of the tool KLEE does not yet handle calls to system directories, i.e., test cases involving system directories, rendering it inapplicable to many cases of the Findutils and Make programs. Also, due to the inherent difficulty and challenge of the test data generation problem, we could not expect, and did not find, that KLEE was able to expose differences between every one of the pairs of original and faulty programs. Therefore, in a second phase we manually augment the test suites, using the bug reports (following the process of Böhme and Roychoudhury [12]), designing 96 additional test cases that reveal (and that fail to reveal) the bugs. We manually generate tests in all situations where there are either fewer than five test cases that reveal a given fault or fewer than five that cover the fault but fail to reveal it, thereby ensuring that all faults have at least five revealing and five non-revealing test cases.

Our experiments were performed at the system level and involved 323,631 mutants, 53,716 branches and 77,151 statements. Every test exercises the entire program as invoked through the command line (rather than unit testing, which is less demanding, but vulnerable to false positives [41]). As a result, both automated and manual test generation were expensive. For example, the machine time that was spent on symbolic execution took approximately 1 day, on average, for each studied bug. All the test execution needed for our experiment took approximately 480 days of computation time to complete (of single-threaded analysis).

Following the recommendations of Xuan *et al.* [42] we refactored[2] the test cases we used to improve the accuracy of our analysis. This practice also helps to elevate the performance of symbolic execution [43]. Finally, each test 'case' is essentially a test *input* that needs a test oracle [44], in order to determine its corresponding output. Fortunately, in our case, we have a reliable and complete test oracle: the output differences between the fixed and the faulty versions.

Overall, the coverage scores levels achieved by the whole test pool are presented in Figure 1.

---

[2]Many test cases form a composition of independent (valid) test cases. We split these tests and formed multiple smaller and independent ones, which preserve their semantics.

D. Tools for Mutation Testing and Coverage Measurement

To conduct our experiment we used several tools in additional to the shadow symbolic execution [36] feature implemented[3] on top of KLEE [13]. To measure statement and branch coverage we used the GNU Gcov utility. To perform mutation, we built a new tool on top of the Frama-C framework [45] as existing tools are not robust and scalable enough to be applied on our subjects. This tool supports both weak and strong mutation, by encoding all the mutants as additional program branches [46]–[48] (for weak mutation testing), and uses program wrappers, similar to those used by shadow symbolic execution, that automatically and precisely record the program outputs (for strong mutation testing).

Our mutation tool reduces the execution cost of strong mutation by checking for strong death, only those mutants that were already weakly killed [9], since any mutant that is not weakly killed by a test case cannot be strongly killed, by definition. We also used the recently-published TCE (Trivial Compiler Equivalence) method [49] to identify and remove strongly equivalent and duplicated mutants, detected by TCE.

We use a timeout in order to avoid the infinite loop problem: a mutant may lead to an infinite loop, which evidently cannot be detectable in general, due to the undecidability of the halting problem. In this way, we are treating (sufficient difference of) execution time as an observable output for the purpose of strong mutation testing. Thus, a mutant is deemed to be distinct from the original program if its execution differs by more than two times the execution of the original program.

The mutation tool includes the (large and varied) set of mutant operators used in previous research [4], [30], [49]. Specifically, we used mutants related to arithmetic, relational, conditional, logical, bitwise, shift, pointers and unary operators. We also used statement deletion, variable and constant replacement.

E. Analyses Performed on the Test Suites

To answer our research questions we performed the following analysis procedure. We constructed a coverage-mutation matrix that records the statements and branches covered and mutants killed by each test case of the test pool.

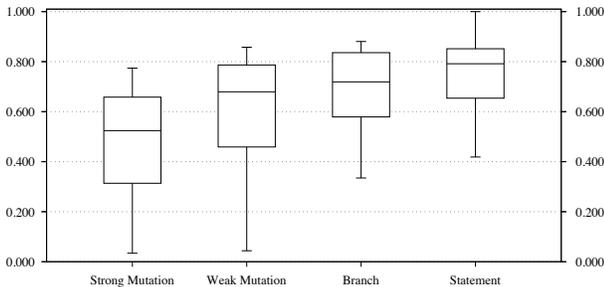[3]http://srg.doc.ic.ac.uk/projects/shadow/

Fig. 1. The test pool with overall coverage score values.

For RQ1 we select arbitrary test sets, execute them in both the fixed (clean) and faulty versions and measure their coverage and mutation scores. We used the Wilcoxon test to compare these values. In order to facilitate inferential statistical testing, we repeat the sampling process 10,000 times so that, for each fault and for each coverage criterion, we perform 10,000 testing experiments, each with a different sampled test suite. The Wilcoxon test is a non-parametric test and thus, it is suitable for samples having unknown distribution [50], [51]. The statistical test allows us to determine whether or not the Null Hypothesis (that there is no difference between the test coverage achieved for the clean and faulty versions of the program) can be rejected. If the Null Hypothesis is rejected, then this provides evidence that the Clean Program Assumption does not hold.

However, statistical significance does not imply practical significance; even when the assumption does not hold, if the effect of assuming it is found to be always small, then the pernicious effects (on previous and potential future experiments) may also be small. Therefore, we also measured the Vargha Delaney effect size $\hat{A}_{12}$ [52], which quantifies the size of the differences (statistical effect size) [50], [51]. The $\hat{A}_{12}$ effect size is simple and intuitive. It measures the probability that values drawn from one set of data will have a different value to those drawn from another. $\hat{A}_{12} = 0.5$ suggests that the data of the two samples tend to be the same. Values of $\hat{A}_{12}$ higher than 0.5 indicate that the first dataset tends to have higher values, while values of $\hat{A}_{12}$ lower than 0.5 indicate that the second data set tends to have higher values.

To study further the differences between the faulty and the fixed program versions, we use the notion of coupling [30], [34], [53]. A fault is coupled with a mutant, statement or branch if every test that kills the mutant (respectively covers the statement or branch) also reveals the fault. Thus, if, for example, a statement is coupled with a fault, then every test set that covers this statement will also reveal this fault. Unfortunately, computing the exact coupling relations is infeasible since this would require exhaustive testing (to consider every possible test set). However, should we find that a fault, f remains uncoupled with all mutants, statements or branches then this provides evidence that the adequacy criterion is not particularly good at uncovering f. Based on the coupled faults we can provide further evidence related to the Clean Program Assumption. If we observe many cases were faults are coupled in one version (either faulty or fixed) while not in the other, then we have evidence against the assumption.

To answer RQ2 and RQ3 we examined the relation between coverage score and fault revelation by selecting test sets of equal size (number of tests). We thus, select 10,000 suites of sizes 2.5% 5%, 7.5%, 10%, 12.5%, and 15% of the test pool (composed of all developer, machine and manually generated test cases). Then, for every score, c_i, in the range [0, maximum recorded score], we estimate the average fault revelation rate for all the tests that have coverage values at least c_i. This rate estimates the probability that an arbitrary c_i%-adequate test suite detects a fault.
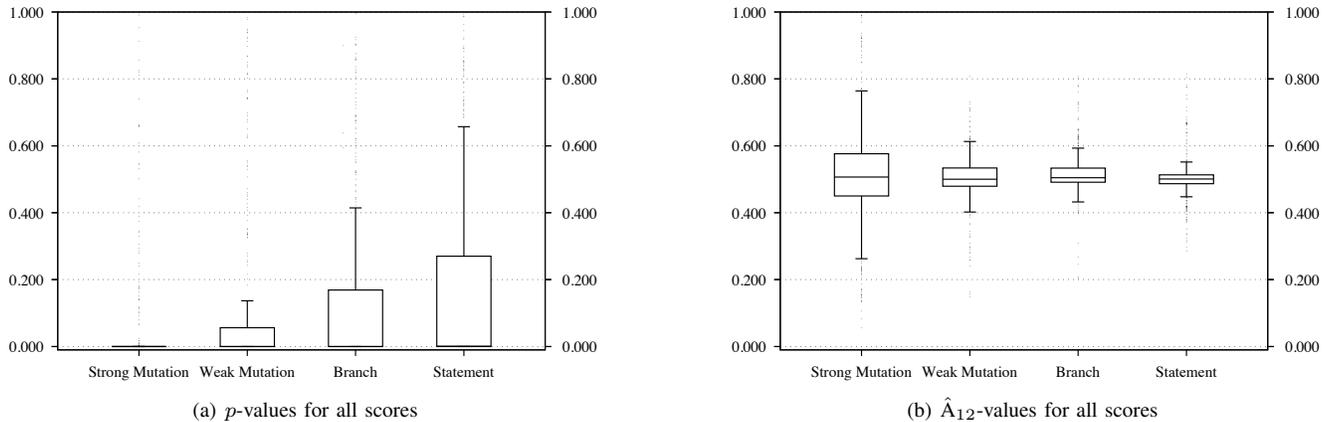
(a) *p*-values for all scores
(b) $\hat{A}_{12}$-values for all scores

Fig. 2. RQ1: Comparing the "Faulty" with the "Clean" ('Fixed') programs. Our results show that there is statistically significant difference between the coverage values attained in the "Faulty" and "Clean" programs (subfigure 2(a)) with effect sizes that can be significant (subfigure 2(b)).

We then compare these fault revelation probabilities for different levels of minimal coverage attainment. Ideally, we would like to control both test size and coverage across the whole spectrum of theoretically possible coverage levels (0-100%). However, since coverage and size are dependent it proved impossible to do this, i.e., large test sizes achieve high coverage, but not lower, while smaller sizes achieve lower coverage but not higher. Therefore, to perform our comparisons we record the highest achieved scores per fault we study. For RQ2 we compared the fault revelation of scores for arbitrary selected test suites with those of the highest 20%, 10%, and 5% coverage attainment (of same size). For RQ3 we compared the fault revelation of the criteria when reaching each level of coverage in turn. To perform the comparisons we used three metrics: a Wilcoxon test to compare whether the observed differences are statistically significant, the Vargha Delaney $\hat{A}_{12}$ for the statistical effect size of the difference and the average fault revelation differences. Finally, to further investigate RQ3, we also compare the number of faults that are coupled with the studied criteria according to our test pool.

V. RESULTS

A. RQ1: Clean Program Assumption

The Clean Program Assumption relies on the belief that the influence of faults on the program behaviour is small. However, white-box adequacy criteria depend on the elements to be tested [31]. Thus, faulty and clean programs have many different test elements simply because their code differs. Unfortunately, applying experiments to the clean version does not tell us what would happen on the program execution (of the same test) of the faulty program versions. Therefore, we seek to investigate the differences in the coverage scores of test suites when applied to the clean and the faulty programs.

The results of our statistical comparison (*p*-values) between the coverage scores obtained from the faulty and clean programs are depicted in Figure 2(a). These data show that all measures differ when applied on the clean rather than the faulty program versions.

These differences are significant (at the 0.05 significance level) for all four criteria and for 86%, 74%, 66% and 60% of the cases for strong mutation, weak mutation, branch and statement coverage respectively. Strong mutation differences are more prevalent than those of the other criteria indicating that the Clean Program Assumption is particularly unreliable for this coverage criterion.

The results related to the effect sizes are depicted in Figure 2(b), revealing that large effect sizes occur on all four criteria. Strong mutation has larger effect sizes than the other criteria, with some extreme cases having very high or low $\hat{A}_{12}$ values.

One interesting observation from the above results is that the faults do not always have the same effect. Sometimes they decrease and sometimes they increase the coverage scores. It is noted that the effect sizes with $\hat{A}_{12}$ values higher than 0.5 denote an increase of the coverage, while below 0.5 denote a decrease. Therefore, the effect of the bias is not consistent and thus, not necessarily predictable.

To further investigate the nature of the differences we measure the couplings between statements, branches and mutants with the faults. Figure 3 presents a Venn diagram with the number of coupled faults in the "Faulty" and the "Clean" versions. We observe that 10, 12, 6, and 4 couplings (represent 16%, 20%, 10% and 7% of the considered faults) are impacted by the version differences when performing statement, branch, weak mutation and strong mutation testing.

We also observe that for statement, branch, weak and strong mutation, 1, 2, 2, and 2 faults are coupled only to test criteria elements on the faulty versions, while 9, 10, 4 and 2 faults only coupled on the clean versions. Interestingly, in the clean versions branch coverage performs better than weak mutation (couples with 37 faults, while weak mutation with 33), while in the faulty version it performs worst (couples with 29, while weak mutation with 31). These data, provide further evidence that results drawn from the two programs can differ in important ways, casting significant doubts on the reliability of the Clean Program Assumption.

Test Criterion	Av Fault Revelation			rand vs. highest 20%			rand vs. highest 10%			rand vs. highest 5%		
	highest 20%	highest 10%	highest 5%	$p-value$	$\hat{A}_{12}$	Av diff	$p-value$	$\hat{A}_{12}$	Av diff	$p-value$	$\hat{A}_{12}$	Av diff
Statement	0.518	0.535	0.553	8.21E-03	0.478	-0.009	1.25E-03	0.457	-0.025	8.97E-04	0.438	-0.043
Branch	0.524	0.542	0.564	1.94E-05	0.467	-0.015	6.24E-05	0.452	-0.033	4.73E-06	0.421	-0.055
Weak Mutation	0.510	0.535	0.555	9.04E-02	0.498	-0.001	2.25E-02	0.472	-0.026	4.74E-04	0.452	-0.045
Strong Mutation	0.565	0.639	0.684	9.39E-06	0.431	-0.057	7.20E-07	0.367	-0.130	1.60E-07	0.340	-0.176

B. RQ2: Fault revelation at higher levels of coverage

The objective of RQ2 is to investigate whether test suites that reach higher levels of coverage (for the same test suite size) also exhibit higher levels of fault revelation. To answer this question we selected 10,000 arbitrary test suites (per fault considered) using uniform sampling so that they all have the same test size. We then compare their fault revelation with that of the tests suites that achieve the highest levels of coverage. Thus, we compare with test suites that lie in the top 5%, 10% and 20% of coverage, to investigate different levels of maximal coverage attainment.

Table III records the results for the controlled test size equal to 7.5% of the test pool, which are representative of those we attained with the other sizes, i.e., 2.5%, 5%, 10%, 12.5% and 15%. Overall, our data demonstrate that all criteria exhibit minor improvement in their fault revelation when considering the threshold of the highest 20% (all $\hat{A}_{12}$ values are above the 0.4). The results are even worse for lower coverage thresholds, i.e., when considering the highest 25%, 30% etc. In practical terms, the fact that the fault revelation differences are small, indicates that test sets having coverage values lying within the highest 20% are those that reveal significantly more faults than arbitrary test sets of the same size.

The surprising finding is that fault revelation improves slightly when test suites achieve the top 20% of the levels of coverage for a given test suite size for all the four criteria. However, for strong mutation, we *do* observe more important differences when the top 10% and the top 5% of coverage are attained. Furthermore, for strong mutation, the average fault revelation rate was 5% higher than the arbitrary test sets (for the highest 20%). This increases to approximately 13% and 18% when considering the test suites that had the highest 10% and 5% coverage attainment.

As can also be seen from the results, confining our attention to only the top 10% (and even the top 5%) levels of coverage attainable for a given test suite size produce only minor improvements in fault revelation for the other three criteria. That is, test suites with the higher 10% and higher 5% of coverage attainment for statement, branch and weak mutation in Table III do not exhibit practical improvements on the fault revelation compared to arbitrary test suites of the same size (the lowest $\hat{A}_{12}$ is 0.421).

(a) Statement
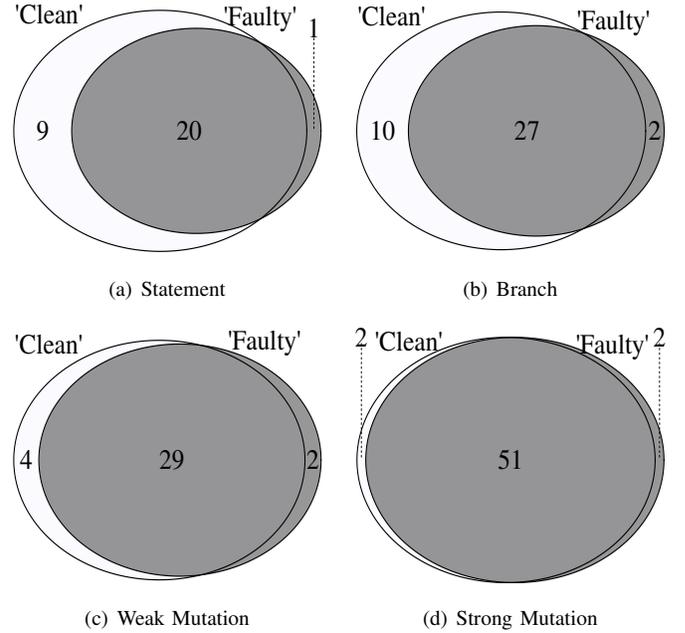
(b) Branch

(c) Weak Mutation

(d) Strong Mutation

Fig. 3. Fault coupling in the 'Faulty' and 'Clean' versions.

By contrast, test suites that lie within the highest 10% and 5% for strong mutation do achieve significantly higher fault revelation than arbitrary test suites of the same size. For both the 10% and 5% thresholds, the differences exhibit relatively strong effect sizes (the Vargha Delaney effect size measures, of 0.340 and 0.367 respectively, which are noticeably lower than 0.50). Furthermore, at the highest 5% the p-value is lowest and the effect size largest.

Taken together, these results provide evidence that test suites that achieve strong mutation coverage have significantly higher fault revelation potential than those that do not, while the improvements for statement, branch and weak mutation are small. This result suggest that coverage should be used only as the starting point of testing and not as a test quality indicator. Finally, we notice that relatively high levels of strong mutation are required (top 10%) for this effect to be observed; below this threshold level, differences between arbitrary and (partially) strong mutation adequate test suites are less important.

Criteria comparison	$p-val$	$\hat{A}_{12}$	Av Fault Revelation Diff
Strong Mut vs Statement	2.61E-05	0.623	0.132
Strong Mut vs Weak Mut	2.23E-07	0.626	0.130
Strong Mut vs Branch	8.16E-04	0.614	0.120
Weak Mut vs Statement	4.65E-01	0.490	0.002
Weak Mut vs Branch	7.23E-02	0.474	-0.010
Branch vs Statement	3.37E-04	0.517	0.012

Fig. 5. Fault coupling between the studied criteria.

C. RQ3: Fault Revelation of Statement, Branch, Weak and Strong Mutation

RQ2 compared arbitrary test suites with higher adequacy test suites of the same size for each coverage criterion. This answered the *within-criteria* question, for each criterion, of whether increasing coverage according to the criterion is beneficial. However, it cannot tell us anything about the differences in the faults a tester would observe *between criteria*, a question to which we now turn.

Table IV reports the differences in pairwise comparisons between the four criteria, for test suites containing 7.5% of the overall pool of test cases available; the same size test suites we used to answer RQ2. Results for other sizes of test suites are similar, but space does not permit us to present them all here. All the differences are statistically significant, the p-values are lower that the 0.05 level, except from the differences of 'weak mutation - Statement' and 'weak mutation - branch'.

The results from this analysis suggest that, in terms of fault revelation, there are differences between statement, branch and the weak mutation, when compared to one another. However, these are small (as measured by the p and $\hat{A}_{12}$ values). The results also indicate that strong mutation significantly outperforms all other criteria, i.e., weak mutation, branch and statement coverage, with fault revelation scores that are, on average, at least 12% higher. Figure 4 visualises these results (fault revelation of the four criteria and randomly selected test suites) and demonstrate the superiority of strong mutation over the other criteria.
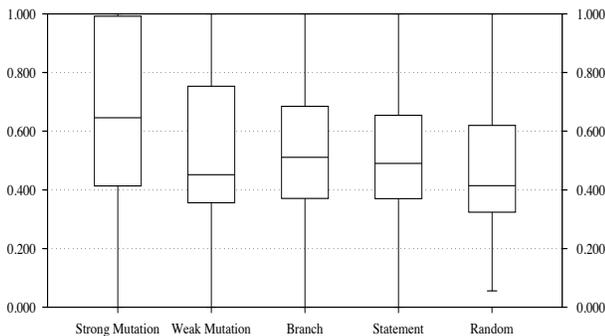

Fig. 4. Fault Revelation of the studied criteria for the highest 5% coverage threshold and test suite size of 7.5% of the test pool.
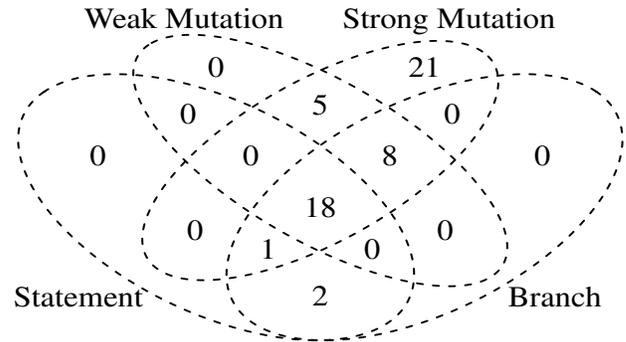
Finally, Figure 5 shows the faults coupled uniquely (and jointly) to each of the four adequacy criteria. This provides another view of the likely behaviour of test suites that target each of these coverage criteria with respect to the real-world faults considered. Each region of the Venn diagram corresponds to an intersection of different coverage criteria, and the number recorded for each region indicates the number of faults coupled by the corresponding intersection of criteria. This allows us to investigate the faults that are uniquely coupled by each criterion, and those coupled jointly by pairs, triples and quadruples of criteria.

In the fault dataset we study there are 61 faults, 6 are not coupled to any of the criteria and 18 are coupled to all four criteria (the quadruple of criteria region depicted in the centre of the Venn diagram). It is interesting that all faults coupled by weak mutation are also coupled by strong mutation, since strong mutation does not theoretically subsume weak mutation. Branch coverage theoretically subsumes statement coverage, but only when 100% of the feasible branches and 100% of the reachable statements are covered; there is no theoretical relationship between partial branch coverage and partial statement coverage. Therefore, it is interesting that, for our partially adequate test suites, all faults are coupled with statement coverage are also coupled with branch coverage. By contrast, 3 faults coupled to branch coverage are not to weak mutation (one of which is coupled to strong mutation), while weak mutation has 5 faults coupled that are uncoupled with branch coverage.

However, differences between statement, branch and weak mutation are relatively small by comparison with the differences we observe between strong mutation and the other three criteria. Indeed, Figure 5 provides further compelling evidence for the superiority of strong mutation testing over the other coverage criteria. As can be seen, 21 faults are uniquely coupled to strong mutation. That is, 21 faults are coupled to strong mutation that are not coupled to *any* of the other criteria (showing that strong mutation uniquely couples to 38% of faults that are coupled to any of the four criteria). By contrast, each of the other three criteria has *no* faults uniquely coupled, and even considering all three together only have two faults that are not coupled to strong mutation. These faults are only coupled to branch and statement coverage.

VI. Threats to Validity

As in every empirical study of programs, generalisation remains an open question, requiring replication studies. We used C utility programs. Programs written in other languages and with different characteristics may behave differently. All four programs we used are "well-specified, well-tested, well-maintained, and widely-used open source programs with standardized program interfaces" [12] with bug reports that are publicly accessible. Our results may generalise to other well-specified, well-tested, well-maintained, and widely-used open source C programs, but we have little evidence to generalise beyond this. Additional work is required to replicate and extend our results, but clearly any future work should either avoid the Clean Program Assumption or first investigate its veracity for the selected pool of subjects.

Another potential threat to the validity of our findings derives from the representativeness of our fault data. We used real faults, isolated by Böhme and Roychoudhury [12] and used by other researchers [36], [54]. Since these faults were found on well-tested widely-used programs, we believe that they are representative of faults that are hard to find, but further research is required to test this belief.

The use of automatically generated and manually augmented test suites also poses a threat to generalisability. While we cannot guarantee the representativeness of this practice, it is desirable in order to perform experiments involving multiple comparisons that use a good mix of tests that reveal (and fail to reveal), the faults studied. We control for test suite size and different levels of achievement of test adequacy, and perform multiple samples of test suites to cater for diversity and variability. Nevertheless, we cannot claim that the test suites we used are necessarily representative of all possible test suites.

We restricted our analysis to the system level testing, since the developers' tests suites were also system level tests and we used a wide set of mutation operators, included in most of the existing mutation testing tools, as suggested by previous research [4], [30], [32], [49]. We view this as an advantage, because, according to Gross *et al.* [41], applying testing at the system level makes robust experimentation that reduces many false alarms raised when applying testing on the unit level, while focusing on a narrower set of mutation operators would tend to increase threats to validity. However, this decision means that our results do not necessarily extend to unit level testing, nor to other sets of mutation operators.

All statements, branches and mutants that cannot be covered (or killed) by any test in our test pool are treated as infeasible (or as equivalent mutants). This is a common practice in this kind of experiment [1], [4], [5], [55], because of the inherent underlying decidability problem. However, it is also a potential limitation of our study, like others. Furthermore, since we observe a 'threshold' behaviour for strong mutation, it could be that similar thresholds apply to statement branch and weak mutation criteria, but these thresholds lie above our ability to generate adequate test suites.

There may be other threats related to the implementation of the tools, our data extraction and the measurements we chose to apply, that we have not considered here. To enable exploration of these potential threats and to facilitate replication and extension of our work, we make available[4] our tools and data.

VII. Conclusion

We present evidence that the Clean Program Assumption does not always hold: there are often statistically significant differences between coverage achieved by a test suite applied to the clean (fixed) program and to each of its faulty versions, and the effect sizes of such differences can be large. These differences are important as they may change the conclusions of experimental studies. According to our data, more faults are coupled with weak mutation than branch testing in the faulty programs but less in the clean ones. This finding means that future empirical studies should either avoid the Clean Program Assumption, or (at least) treat it as a potential threat to the validity of their findings. The unreliability of the Clean Program Assumption motivated us to reconsider the relationship between four popular test adequacy criteria and their fault revelation. We thus reported empirical results based on an experimental methodology that benefits from enhanced robustness (by avoiding the Clean Program Assumption).

In this study, we provide evidence to support the claim that strong mutation testing yields high fault revelation, while statement, branch and weak mutation testing enjoy no such fault revealing ability. Our findings also revealed that only the highest levels of strong mutation coverage attainment have strong fault-revealing potential. An important consequence of this observation is that testers will need to have first achieved a threshold level of coverage before they can expect to receive the benefit of increasing fault revelation with further increases in coverage.

Future work includes studies and experiments aiming at increasing the understanding of these fundamental aspects of software testing. An emerging question regards the optimal use of mutants when comparing testing methods, i.e., whether methods should be applied on the original (clean) or on the mutant versions of the programs. Similarly, the relation of specific kinds of mutants, such as the subsuming [56] and hard to kill [57] ones, with real faults and their actual contribution within the testing process form other important aspects that we plan to investigate.

---

[4]https://sites.google.com/site/mikepapadakis/faults-mutants

REFERENCES

[1] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, 2013, pp. 302–313. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483769

[2] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 72–82. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568278

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 435–445. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568271

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.83

[5] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*, 1998, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/288195.288298

[6] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[7] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing (keynote)," in 8th *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015. [Online]. Available: http://dx.doi.org/10.1109/ICST.2015.7102580

[8] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.93

[9] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Mutation 2010*, 2010, pp. 90–99. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2010.50

[10] Radio Technical Commission for Aeronautics, "RTCA DO178-B Software considerations in airborne systems and equipment certification," 1992.

[11] S. C. Reid, "The software testing standard — how you can use it," in 3rd *European Conference on Software Testing, Analysis and Review (EuroSTAR '95)*, London, Nov. 1995.

[12] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, 2014, pp. 105–115. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628058

[13] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[14] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: Haven't we solved this problem yet?" in 4th *Testing Academia and Industry Conference — Practice And Research Techniques*, Windsor, UK, 4th–6th September 2009, pp. 95–104. [Online]. Available: http://dx.doi.org/10.1109/TAICPART.2009.15

[15] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.061

[16] L. C. Briand and D. Pfahl, "Using simulation for assessing the real impact of test-coverage on defect-coverage," *IEEE Trans. Reliability*, vol. 49, no. 1, pp. 60–70, 2000. [Online]. Available: http://dx.doi.org/10.1109/24.855537

[17] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572280

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, p. 22, 2015. [Online]. Available: http://doi.acm.org/10.1145/2660767

[19] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[20] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 156–173, 1975. [Online]. Available: http://dx.doi.org/10.1109/TSE.1975.6312836

[21] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds., Los Alamitos, California, USA, 2007, this volume. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.25

[22] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Symposium on Testing, Analysis, and Verification*, 1991, pp. 154–164. [Online]. Available: http://doi.acm.org/10.1145/120807.120821

[23] ——, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Trans. Software Eng.*, vol. 19, no. 8, pp. 774–787, 1993. [Online]. Available: http://dx.doi.org/10.1109/32.238581

[24] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Softw., Pract. Exper.*, vol. 26, no. 2, pp. 165–176, 1996. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2⟨165::AID-SPE5⟩3.0.CO;2-K

[25] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(96)00154-9

[26] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, 2009, pp. 220–229. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2009.30

[27] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the number and nature of faults found by random testing," *Softw. Test., Verif. Reliab.*, vol. 21, no. 1, pp. 3–28, 2011. [Online]. Available: http://dx.doi.org/10.1002/stvr.415

[28] Y. Wei, B. Meyer, and M. Oriol, *Is Branch Coverage a Good Measure of Testing Effectiveness?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 194–212. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25231-0_5

[29] M. M. Hassan and J. H. Andrews, "Comparing multi-point stride coverage and dataflow coverage," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 172–181. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606563

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "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, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 654–665. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635929

[31] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[32] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[33] K. Androutsopoulos, D. Clark, H. Dan, M. Harman, and R. Hierons, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in 36th *International Conference on Software Engineering (ICSE 2014)*, Hyderabad, India, June 2014, pp. 573–583. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568314

[34] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, 1992. [Online]. Available: http://doi.acm.org/10.1145/125489.125473

[35] R. Gopinath, C. Jensen, and A. Groce, "The theory of composite faults," in *10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-18, 2017*, 2017.

[36] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 1181–1192. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884845

[37] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 191–200. [Online]. Available: http://portal.acm.org/citation.cfm?id=257734.257766

[38] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10664-005-3861-2

[39] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[40] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 549–552. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287707

[41] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336762

[42] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-refactoring: Automatic test code refactoring to improve dynamic analysis," *Information & Software Technology*, vol. 76, pp. 65–80, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2016.04.016

[43] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: delta debugging, even without bugs," *Softw. Test., Verif. Reliab.*, vol. 26, no. 1, pp. 40–68, 2016. [Online]. Available: http://dx.doi.org/10.1002/stvr.1574

[44] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2372785

[45] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0326-7

[46] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, pp. 691–723, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11219-011-9142-y

[47] ——, "Mutation based test case generation via a path selection strategy," *Information & Software Technology*, vol. 54, no. 9, pp. 915–932, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2012.02.004

[48] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICST.2015.7102607

[49] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 936–946. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.103

[50] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*, 2011, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985795

[51] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Springer, 2000.

[52] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Jrnl. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000. [Online]. Available: http://dx.doi.org/10.3102/10769986025002101

[53] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1509

[54] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 471–482. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.65

[55] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 92–102. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693070

[56] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 354–365. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931040

[57] W. Visser, "What makes killing a mutant hard," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 39–44. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970345