Coverage and Its Discontents

Alex Groce Mohammad Amin Alipour Rahul Gopinath
Oregon State University
agroce@gmail.com, aminalipour@gmail.com, gopinath@eecs.oregonstate.edu

Abstract

Everyone wants to know one thing about a test suite: will it detect enough bugs? Unfortunately, in most settings that matter, answering this question directly is impractical or impossible. Software engineers and researchers therefore tend to rely on various measures of code coverage (where mutation testing is considered a form of syntactic coverage). A long line of academic research efforts have attempted to determine whether relying on coverage as a substitute for fault detection is a reasonable solution to the problems of test suite evaluation. This essay argues that the profusion of coverage-related literature is in part a sign of an underlying uncertainty as to what exactly it is that measuring coverage should achieve, as well as how we would know if it can, in fact, achieve it. We propose some solutions and mitigations, but the primary focus of this essay is to clarify the state of current confusions regarding this key problem for effective software testing.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Experimentation, Measurement, Verification

Keywords testing, coverage, evaluation

1. Introduction

A specter is haunting software testing — the specter of not knowing whether our test suites are effective, or how to make them more effective. A few of the questions researchers and even many practitioners would like to be able to answer are:

1. Can I stop testing? Is this suite good enough that I can assume any undetected defects are either very few in number or are so low-probability that they are unlikely to cause trouble? Similarly, will this suite likely detect future defects introduced into this program?
2. Which of these two test suites will find more faults?
3. Which of these two test suite generation methods produces better suites, in terms of detecting faults?
4. In what order should I run the tests in this suite (or which tests should I run), if I want to maximize the speed with which faults are detected?
5. What tests should I add to this test suite, to improve its ability to detect faults?

Unfortunately, the only obvious way to answer all of these questions is to have a list of all faults in our software programs[1]! In the presence of such a list, of course, the entire problem of testing is avoided, which means that these pressing questions must somehow be "answered" without the needed information. Producing such a list of faults, in practice, is difficult even if we have a complete (with respect to faults) set of test cases for a program! Detecting faults not only requires executing the program such that the fault causes a problem, but some method for detecting that the test case has "done the wrong thing" and exposed a fault. This is known as the *oracle problem*: an oracle is a function that, given a test case execution, determines if it passes or fails [36]. Building an oracle with respect to a given kind of failure can be trivial (e.g. for crashes and infinite loops) or profoundly difficult (e.g. for full, functional correctness properties — the easiest way to determine if a file system functions properly is to test against another, correct, file system); in some cases, the oracle cannot be automated, and requires human intervention [30]. Even with a perfect oracle, determining the list of faults from a set of failed tests is a very hard problem [11].

The most frequently proposed (and adopted) solution is to use some measure of (code) coverage, which Beizer [7] defines as "any metric of completeness with respect to a test selection criterion" and notes usually means "branch or statement coverage." Statement coverage measures how many of the statements in a program have been executed by a test suite, and branch coverage measures how many of

Onward! 2014, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-3210-1/14/10…$15.00.
http://dx.doi.org/10.1145/2661136.2661157

[1] Actually, even given this unlikely boon, answering the third and fifth questions is still difficult.

the branches in a program have been taken by a test suite. Statement and branch coverage are easily measured in most widely used languages, and tools for these two coverages are included in the basic tool set for many development environments, whether commercial, open source, or more research-oriented (e.g. Visual Studio, GCC, LLVM, and even the Glasgow Haskell Compiler). One great advantage of coverage measures over other possible methods for estimating the fault detection capabilities of a test suite is that coverage can be used even when all tests in the suite pass. This is critical, because it is most difficult to assess a suite's value when it no longer detects any bugs [25]: is the suite too weak, or is the software actually mostly correct? In testing research, coverage is also highly useful for this case — using coverage allows researchers to evaluate the quality of test suites for subject programs that lack faults (or where the effort to build an oracle to detect present faults is too onerous).

The academic literature has proposed a very large set of plausible coverage measures, but only statement and branch coverage are in widespread use by practitioners, and available for most programming languages. In fact, from the perspective of "real-world" development, it is almost true that statement and branch coverage are the only relevant coverage measures. The only other coverage with significant real-world adoption is MC/DC coverage, which must be satisfied for software to obtain Federal Aviation Administration approval for airborne computer software [52], but no widely used development environments support its measurement[2]. Fortunately, there is at least some evidence that statement and branch coverage are not only the most readily available measures, they also are possibly the most effective [18, 19, 22].

That a coverage requirement is specified by the United States government for certain types of critical software, and that tools to measure common coverages are standard in common development environments demonstrates that the idea of coverage as a way to measure test suite effectiveness is not limited to researchers. Many companies make some form of code coverage a requirement in development, and practical, developer-oriented books covering testing almost always discuss statement and branch coverage [41, 43]. Such discussions often include a warning to not use coverage as a mindless criterion for when to stop testing [56]. It nonetheless seems likely that in the future, as testing becomes more automated, more frequently outsourced, and more often offered as a service, coverage will be used as a measure of testing effectiveness even more frequently.

Code coverage is also very widely used in software testing research. Here, in principle, using real faults to compare suites is possible. However, in practice researchers discover a series of obstacles to this ideal approach. First, many programs have few enough (discoverable) faults that obtaining statistical significance in results is difficult [6, 13]. Second, many experiments are performed, for purposes of comparing with related literature, on a set of standard benchmarks [50], and the benchmarks typically have only seeded faults, not real faults. Finally[3], for larger real programs, determining how many different faults a test suite detects can be extremely difficult, and relies on generating a valid oracle for the program. It has become common practice, as a result, for papers in the field to report comparisons of testing techniques that largely or partially [21, 28, 55, 58] focus on comparing the *coverage* produced by the suites [18, 19]. Some papers that rely largely on comparisons of coverage (e.g. the work of Visser et al. [58]) have become very widely cited standards in the field. Branch coverage itself (or path coverage focused on maximizing branch coverage) is also often used as the *actual goal* of automated testing systems, either as a fitness measure in evolutionary/search-based/machine-learning based testing [3, 17, 27, 44] or in symbolic execution efforts [20, 54, 62].

The widespread use of code coverage in both practice and research, therefore, raises a question: is code coverage actually useful as a replacement for measuring real fault detection? Does using coverage in place of faults provide benefits, or is it simply a case of "we have to use something, and this is the only something we have"? Most experienced testers can immediately answer that measuring code coverage is *not* a completely adequate replacement for measuring fault detection; we all know faults that cannot be detected by suites with excellent code coverage. In fact, almost no researchers or experienced testers would affirm the claim "Code coverage is a highly accurate guide to test suite quality; bad suites invariably have poor code coverage." The usual mantra is "coverage is necessary but not sufficient to expose bugs." Nonetheless, both practitioners and researchers frequently use coverage for purposes that far exceed a simple heuristic to detect glaring inadequacies in a suite — as the (perhaps troubling — see the work of Staats et. al [56]) example of MC/DC coverage shows. Is this a good way for scientists and engineers to proceed?

2. Justifying the Use of Coverage

Fortunately, in order for coverage to be valuable in most of the ways we currently assume it to be, it is not essential that coverage have a perfect correspondence to fault detection. Even a moderate *statistical* correlation between some useable coverage measure and actual fault detection, if widespread enough across programs, suites, and fault sets, would justify the use of coverage for at least the first two questions. The notion is fundamentally statistical in that the

[2] One Java coverage tool, CodeCover [1], does offer a measure that subsumes MC/DC coverage.

[3] Actually, there is one additional reason: testing bleeding edge software to find unknown faults is perhaps the most convincing way to reach a development audience, contributes to improving software quality, and is more interesting for researchers than statistically sound "dead horse flogging" [49].

particular faults of any program are largely an accident of design and implementation, so no *generalized* measurement of how a test suite explores program behavior can be expected to have a very precise correspondence to the accidental faults of that program. However, given the pressing need to have some way of at least roughly evaluating the utility of test suites, and some testing goals more amenable to both human effort and automation than "find all faults," a "moderate" correlation would be sufficient. Ignoring the complexities of different statistical measures, a moderate correlation (e.g., a 0.30 score for Kendall's τ correlation) often can be understood as implying a fairly strong statement, such as that if test suite A has a higher coverage measure than test suite B, it is *twice as likely* to be better at fault detection than B than it is to be worse than B at detecting faults. As a result, increasing coverage would generally increase fault detection if this were true. This leads to the development of what this paper will refer to as the Strong[4] Coverage Hypothesis (which can be specialized to a desired form of coverage):

Definition 1. The Strong Coverage Hypothesis (SCH)*: For the population of realistic software systems, test suites produced by human efforts or automated testing methods, and realistic faults, there is at least a moderate statistical correlation between the level of coverage a suite achieves and its level of fault detection. Moreover, this correlation is not the result of some trivial confounding factor that could be used in place of coverage, such as suite size.*

The SCH explicitly claims that coverage is meaningful *in and of itself*, not simply as a byproduct of some other easily measured aspect of a test suite. For example, we expect that very small test suites obtain poor coverage and poor fault detection, and large test suites do well at both coverage and fault detection. Harder et. al warned researchers more than 10 years ago that comparisons of testing methods should be careful to make sure a "better" testing method is not simply forcing the user to spend more computational effort and produce bigger test suites [35]. Both practitioners and researchers care not only about suite effectiveness but suite *efficiency* [32]: if the only way to achieve good fault detection is to use unreasonably large test suites, testing is doomed to failure in most cases. Furthermore, if size is the primary reason for coverage's correlation, it is better (and easier) to just measure suite size in the first place. The SCH proposes that coverage provides useful information not otherwise easily available — given two suites of the same size, the practitioner and the researcher would like to prefer the one with higher coverage, with confidence.

Coverage might also be useful in some specialized settings, or for some of the more specialized questions (prioritizing or augmenting test suites) even if the SCH does not hold. Fundamentally, however, if there is a general positive statistical relationship between a coverage and fault detection, it is reasonable to expect that coverage to be at least somewhat useful for *all* the questions above. How would we go about determining if the SCH is true? Experiments to support it would need to satisfy three requirements:

1. **Large numbers of software systems of different types:** Basing results on only a handful of systems, or systems by a small set of developers, or only a single kind of program (e.g. Unix utilities, GUIs, numerical programs) does not support the SCH. The best basis for the SCH is examination of a large number of randomly (not opportunistically — programs we happened to already be studying) selected software systems with different designs, implementation languages, and specifications. This reduces the danger that there are general classes of software where coverage is not useful, vs. the statistical possibility left open by the SCH that for the occasional program, coverage may be less useful.
2. **Large numbers of test suites of different sizes produced by different methods:** Similarly, the SCH claims that the correlation between coverage and fault detection should not depend on a certain kind of testing method. Therefore experimental evidence should use good test suites and bad test suites, test suites produced by human effort, test suites produced by capturing user inputs, test suites produced by a variety of automated methods (e.g., random testing [33], evolutionary/search-based techniques [17], symbolic execution [20], and model checking [57]), and as many other variations as possible. The SCH intentionally limits correlation to suites someone might actually want to produce and evaluate, so random subsets of tests from many methods are not ideal for evaluation[5], though if there is a good correlation across random subsets of real suites, that is a good sign the SCH holds. Measuring correlation either across suite pools all of the same size or using statistical methods to take size into account is also important. For the first method, using different sizes is important: perhaps coverage is correlated with fault detection in small suites, but is not useful for more realistically sized suites.
3. **Large sets of real faults:** To establish statistical correlation with fault detection, experiments need to actually count faults detected by suites. When no one version of a software system has enough faults to provide statistical support, using many versions (from source repositories) can provide enough faults to measure correlation effectively. Real faults may not resemble seeded faults, so experiments should be based on real faults introduced

[4] To be honest, our "Strong" Coverage Hypothesis is not especially strong, in that it only concerns a limited set of suites, making no claims about the population of all programs or all possible test suites; it's not clear that anyone would believe the Very Strong Coverage Hypothesis, however, or that it could be effectively empirically investigated.

[5] Random subsets are not ideal, for practical purposes; they are widely used, however, because they are easy to obtain [19, 38].

during development, not abstract ideas of what faults "should" look like.

A very large body of literature exists on the topic of the effectiveness of coverage in testing [9, 12, 15, 16, 31, 37, 59, 60] taking a complex (and sometimes confusing) variety of approaches to the question. The hope that something like the SCH is true (or the fear that it is not true) has inspired a growing body of somewhat more focused recent experimental literature [18, 19, 22, 38, 46]. Some of this work [18, 19] is largely motivated by the use of coverage in testing research; other papers are more focused on the use of coverage by practitioners [22]. All of these papers can be understood as attempting to generate evidence for or against some version of the SCH, by choosing some set of programs and suites and measuring statistical correlation. One thing recent papers agree on is that previous work exploring variations of the SCH is inadequate, either focusing on a very small set of programs, using too few test suites, or some combination of these problems (along with a few other methodological objections or at least proposed improvements). That this work, using modern (automated) testing methods, larger bodies of programs (in multiple languages), open source repositories, and better statistical methods, is appearing is a good thing; we need to know if the SCH (or something like it) is true, and we *really* need to know if the SCH is false[6]!

Unfortunately, most of this work doesn't really shed much light on the SCH at all, though at first glance it would appear to do so. Papers by Gligoric et al. and Gopinath et al. [18, 19, 22], appearing in the last year suggest the SCH is probably true; another paper [38] suggests the SCH is not true, once the size of test suites is taken into account[7]. The contradictory results are not the most important problem, however. The problem is that the most recent papers, in order to scale to more programs and more suites, all rely on correlating code coverage to something other than fault detection.

3. Mutation is the Elephant in the Room

The recent papers most directly addressing the SCH [18, 19, 22, 38], and numerous other papers in the literature of code coverage, replace fault detection in the SCH with the use of *mutation analysis*. Mutation analysis [10, 34], also known as mutation testing or syntactic coverage [4], involves simulating faults in a program by making a large number of small syntactic changes to the program. Given a program and a test suite, mutation analysis essentially reports how many of the program's mutants a test suite can distinguish from the original program. The theoretical logic behind mutation testing is that programmers write programs that are close to correct (this is known as the Competent Programmer Hypothesis), and that detecting simple mutants often implies detecting more complex compositions of mutants (known as the Coupling Effect). The empirical evidence for both hypotheses is interesting but not completely compelling. Most of the justification for using mutation testing to explore the effectiveness of code coverage, in practice, comes from assuming a variation of the Strong Coverage Hypothesis where "coverage" is replaced by mutation analysis: detecting more mutants implies detecting more faults, usually. Unfortunately, for the most part, the objections raised against older experiments supporting the SCH apply (or apply even more strongly) to experiments that explore the SCH for mutation analysis, as noted in recent work [23, 40].

We are beginning to assemble interesting evidence about coverage measures, but unfortunately the evidence thus far assembled mostly relies on an also unproven replacement for fault detection. Even if an SCH holds for mutation analysis, there is reason to suspect that the correlation between code coverage and mutation analysis might be stronger than the correlation between coverage and fault detection. Mutants are spread evenly throughout a program, with most methods producing at least one mutant per statement. This close connection of statement coverage to mutation analysis may be the cause of its better correlation with mutation kills than some other measures [22]. To detect any mutant of a line of code, you have to cover it; statement coverage is more able to "count" missed lines than branch coverage. The variance in lines under branches is extreme: in some cases 80% of statements are under 20% of branches [48]. No such argument obviously holds for faults; how faults are distributed throughout a program is a subtle and much discussed question, without any clear, generalizable, answers yet appearing to our knowledge. Even if we had a rigorous and sufficiently large study showing the SCH holds between a code coverage measure and mutation analysis, and another study showing that the SCH holds between mutation analysis and fault detection, the relationship between code coverage and fault detection might be much weaker, to the point that it is unsafe to rely upon. Work is beginning to appear examining the empirical effectiveness of mutation testing [40] and the Competent Programmer Hypothesis [23], but it must be considered preliminary at best.

4. How Can Users of Code Coverage Sleep Well at Night Again?

There is, first, a pressing need for serious large scale evaluation of the connection between mutation analysis and fault detection. For example, a recent FSE paper examining the topic [40] is a good beginning to this topic, but is not sufficient. The paper [40] only considers 5 programs and their suites, and primarily shows that test suites detecting more faults also detect more mutants. It unfortunately is much harder to show that *suites detecting more mutants also detect*

[6] Full disclosure: the authors of this essay wrote some of these papers, so we're unlikely to be against this work.

[7] The reader who proceeds from this paper to the coverage literature will find that the story of the relationship between coverage and suite size is long and complex, with no clear, consistent findings.

more faults, which is required for the SCH. Performing large scale experiments connecting mutation analysis with fault detection is daunting, both in terms of computational needs (running mutation analysis is expensive) and, perhaps more importantly, in terms of the difficulty of determining which tests detect which faults for many large programs. The studies of code coverage use mutation analysis in place of faults because it allows them to avoid this problem (and gives results for programs with no or few known faults)! Scaling to a study of the SCH for mutation over many real programs, selected without bias, which is the ideal minimum for good support, is going to require heroic efforts. Even examining a much simpler question, the syntactic similarity of *detected and corrected* faults to mutants, is quite difficult [23]. If the correlation between mutation analysis and fault detection turns out to be *extremely* strong, the experiments of recent papers can arguably be used to support the SCH, and practice and research can continue happily to rely on coverage as a suite evaluation method.

The correlation may prove weak or moderate, however (or non-existent, though in practice this would probably suggest the SCH does not hold for branch or statement coverage, either). In that case, the difficult work of producing serious experiments satisfying all three requirements (many programs, many suites, and many *real* faults) for coverage measures, not just mutants, needs to be done. It may be a long time before truly satisfactory evidence appears. In the meantime, it would be nice to have a better understanding of the prospects of the SCH. Is there an alternative to waiting for some heroic researchers (or hopefully, multiple independent teams of heroic researchers) to satisfy our need for solid examination of the SCH?

We propose that researchers (and interested practitioners) can begin to build a case for or against the SCH without heroic effort simply by reporting more data from their current testing efforts (see `testcoverage.eecs.oregonstate.edu`). Whenever a research effort involves a program with many real faults, researchers should report code coverage for suites used in their work, even if these results are not essential to the other goals of the research. Industrial or open source testing efforts comfortable with sharing some data may also wish to contribute, in hopes of benefiting from the results. The effort required in most languages is minimal, and many papers in the testing literature already produce multiple test suites and evaluate these suites to support novel testing methods (any experiment examining a non-deterministic testing technique, such as random testing or search-based techniques, will need to generate many suites to achieve statistical confidence). Accumulating many such **(coverage, fault detection)** pairs can at least give some idea as to whether the SCH is likely to be true. Moreover, more specialized versions of the SCH in domains where large-scale automated testing is frequently applied (e.g. compilers [11] or systems utilities [45]) may be useful to practitioners and researchers in these areas.

Even very indirect evidence can shed light on the SCH: e.g., we know that for some programs, reducing the size of test cases (not suites) but maintaining code coverage does not seem to reduce fault detection [29]. This could be true even if the SCH is false, but it supports a basic connection between coverage and fault detection. Similarly, there is experimental evidence for the use of coverage in test suite prioritization and selection, with varying implications for the SCH [8, 14, 51, 61].

5. A Sketch of a Data Set

In order to encourage other researchers to report these "side" statistics about coverage and fault detection, even if the researchers are not interested in producing research on the SCH themselves, we show how easy it can be, given the infrastructure required for performing testing experiments in general.

Figure 1 shows the relationship between branch coverage and fault detection for randomly generated test suites for Mozilla's SpiderMonkey JavaScript engine, for release versions 1.6 and version 1.7. These 192 suites for each version were produced using the `jsfunfuzz` random tester, which has resulted in the discovery of more than 1,700 previously unknown faults in JavaScript engines [53]. Each suite is produced by running `jsfunfuzz` for 30 minutes, removing the possibility of suite size (if measured using computational effort, which seems the most reasonable way to measure it) as a confounding factor. The number of faults detected is estimated (with relatively good accuracy, we believe) using a binary search through the source code repository to find the change that "fixes" each failing test [11, 29].

The graphs show that coverage and fault detection vary widely and independently, but there does appear to be some relationship between the two factors. Applying Pearson's statistical measure for correlation, we see there is a moderate positive correlation ($r = 0.31$) between branch coverage and fault detection for the 1.6 suites, and a slightly stronger positive correlation ($r = 0.36$) for the 1.7 suites. Pearson's correlation measures not only that the two values are related, but that they have a *linear* relationship. For the 1.6 suites, statement coverage has better correlation ($r = 0.34$), but for 1.7 statement coverage is not quite weakly correlated ($r = 0.18$). Switching from Pearson's correlation to Kendall τ_b we see the correlations shown in Table 5, as computed by the R statistical package. While Pearson's correlation measures linear relationships, τ_b simply measures the degree to which suites with higher coverage are also likely to have higher fault detection, which is probably most important. The table helps us understand the results a bit better; while there is moderate correlation between coverage(s) and fault detection, there is a much better correlation between fault detection and either the number of tests executed

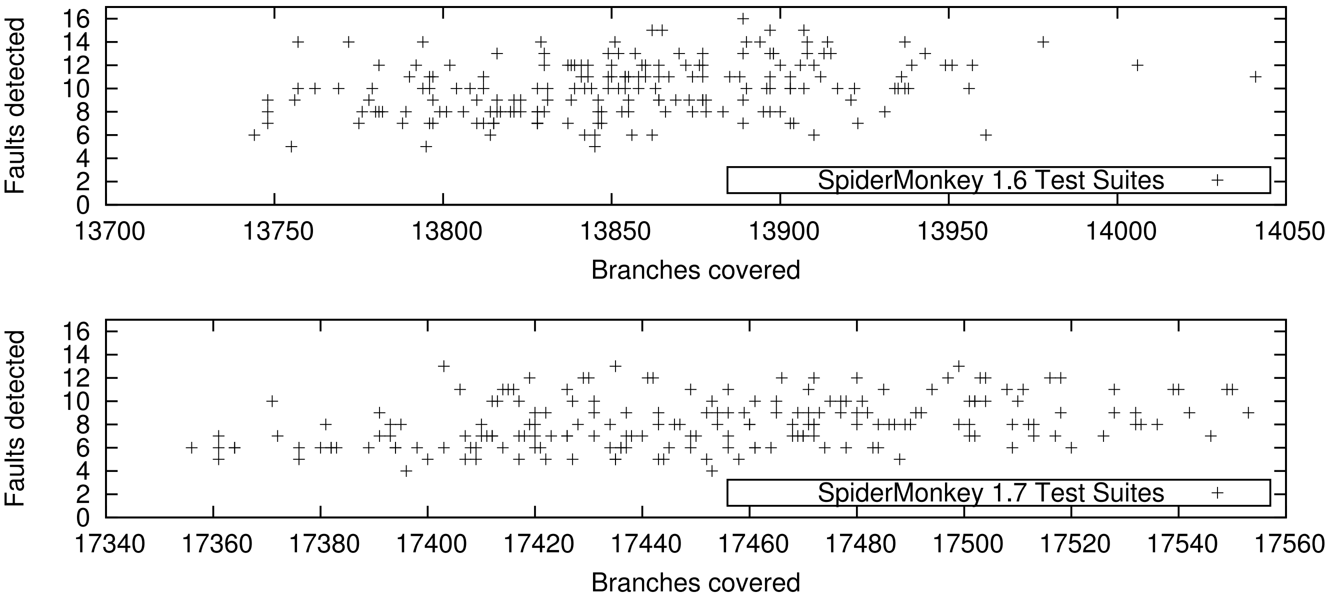

Figure 1. SpiderMonkey 1.6 and 1.7 Test Suites

Table 1. τ-b Correlation for Spidermonkey Test Suites
bcov = branch coverage; scov = statement coverage; fcov = function coverage; #t = number of tests; #fail = number of failing tests; #faults = number of detected faults

SpiderMonkey Version 1.6						
	bcov	scov	fcov	#t	#fail	#faults
bcov	1.0000000	0.7271282	0.4921943	0.2415809	0.1791301	0.2341907
scov	0.7271282	1.0000000	0.6390468	0.2809492	0.1773752	0.2668531
fcov	0.4921943	0.6390468	1.0000000	0.3439814	0.2421777	0.2801842
#t	0.2415809	0.2809492	0.3439814	1.0000000	0.5161900	0.4648379
#fail	0.1791301	0.1773752	0.2421777	0.5161900	1.0000000	0.4505282
#faults	0.2341907	0.2668531	0.2801842	0.4648379	0.4505282	1.0000000
SpiderMonkey Version 1.7						
	bcov	scov	fcov	#t	#fail	#faults
bcov	1.0000000	0.6250377	0.32061158	0.34191047	0.2384296	0.27300589
scov	0.6250377	1.0000000	0.46445435	0.17936413	0.1542907	0.15160428
fcov	0.3206116	0.4644543	1.00000000	0.06873412	0.0888408	0.06043132
#t	0.3419105	0.1793641	0.06873412	1.00000000	0.5317346	0.54171101
#fail	0.2384296	0.1542907	0.08884080	0.53173458	1.0000000	0.58647031
#faults	0.2730059	0.1516043	0.06043132	0.54171101	0.5864703	1.00000000

in 30 minutes or the number of tests that fail. Perhaps our conclusion that "size" cannot be a confounding factor here is mistaken? In fact, it turns out the correlation is valid but somewhat irrelevant; because jsfunfuzz terminates a test execution as soon as a failure occurs, suites with more failures will execute more tests (the strongest single correlation in either data set, other than that between statement and branch coverage). That the number of failed tests is a better predictor than coverage should surprise no one. Counting failed tests is not a good replacement for measuring code coverage, however: recall that coverage is most useful when a suite is producing no failures, but it is not known whether this is due to the quality of the code or the lack of quality in the suite. It seems reasonable to say that the SpiderMonkey data forms a small piece of evidence in favor of the SCH.

At this point a procedure for moving towards solid experimental support for coverage as valid seems to appear: assemble enough significant programs with long revision histories, plenty of real faults, and effective automated testing procedures that can produce large numbers of equal-compute-time suites. If the results consistently show respectable correlation between coverage and fault detection, across a variety of types of program, testing methods, and testing budgets, it is safe to assume that coverage is an effective predictor of fault detection, and put the specter to rest. Certain problems will remain, of course. The evidence will support coverage as effective for evaluating generated test suites, but human-produced suites may systematically differ from automated testing. The correlations will probably not be as high as shown in experiments with mutants, since real faults are less systematically distributed. Real fault data can give us insights into detected and fixed faults, but leaves open the possibility that some important faults are undetected by most current testing practices. Nonetheless, a series of similar experiments would considerably ease the consciences of testing researchers.

5.1 No Simple Answers

Unfortunately, the JavaScript story above is missing a critical detail. Half of the 1.6 test suites and half of the 1.7 test suites were generated with an unmodified version of `jsfunfuzz`. The other half were generated using the swarm testing method [28], which has the same overall random test domain, but modifies the probability distribution in each test in a way that typically improves both coverage and fault detection. Experimental validation and improvement of swarm testing was the reason we produced this data in the first place! When we split the 1.6 and 1.7 suites into subpopulations of swarm suites and default (the original version of `jsfunfuzz`) suites, the relationship between coverage and fault detection becomes much less apparent (Figures 2 and 3). Pearson's measure now shows essentially no correlation between branch or statement coverage for either swarm or default suites, for SpiderMonkey 1.6 (-0.03 $< r <$ 0.05). The same is true of swarm tests for SpiderMonkey 1.7 (where both branch and statement have small negative r). However, default suites for SpiderMonkey 1.7 still show a modest positive correlation between coverage (either branch or statement, with $r = 0.29$) and fault detection! Table 5.1 shows Kendall $\tau - b$ correlations for the test suites when the population is split by test generation method (values are now rounded in order to fit in the table). If we add a binary value for whether the suite is a swarm or default suite to the original all-suites data, that turns out to be the variable with the highest correlation to the suite's coverage measures and fault detection power.

This shows a potential problem in any experiments intended to show that coverage (of any kind) has a useful relationship with coverage. Namely, the relationship may be one where some common factor (here, whether the test was produced using swarm techniques) produces both better coverages and better fault detection. The correlation may disappear in populations of tests that lack such an underlying cause! Is this a problem?

A confounding factor such as suite size makes coverage fairly useless. It is easier to measure suite size than coverage. What about a confounding factor like "these tests were produced by a better testing process?" This is definitely a negative data point for the SCH, but our interest is in practical use of coverage, not in the SCH itself. Testing is inherently a field interested in practice, even in its research aspects. Even if coverage is only a sign of some underlying factor, if it is a consistent sign of such factors when they exist, coverage is useful for testing and testing research. The reason coverage is not very helpful for three of the four data sets produced when suites are split by generation method is that the differences in these suites are *entirely* attributable to the accidents of a pseudo-random number generator. That fault detection and coverage are essentially randomly distributed in that case isn't really surprising, and it is not clear why we would want to select the "best" of these basically interchangeable test suites. Choosing between swarm and default testing, on the other hand, is a highly desirable goal, and here measuring coverage seems to provide reasonable prediction, with little computational effort, of the average fault detection capability of swarm suites vs. default suites. Similarly, if two manual efforts to produce effective test suites do not differ in some interesting way, other than the accidents of which tests are selected — if the testers are equally skilled, resource-enabled, and share the same testing goals, we probably don't really care if we can cheaply distinguish between the suites. The only source of difference is "luck." Expecting coverage to correlate in such homogeneous populations seems like too strict a requirement. We therefore propose the **Weak Coverage Hypothesis**:

Definition 2. The Weak Coverage Hypothesis (WCH)*:* *For the population of realistic software systems, test suites produced by* different *human efforts or* different *automated testing methods, and realistic faults, there is at least a mod-*

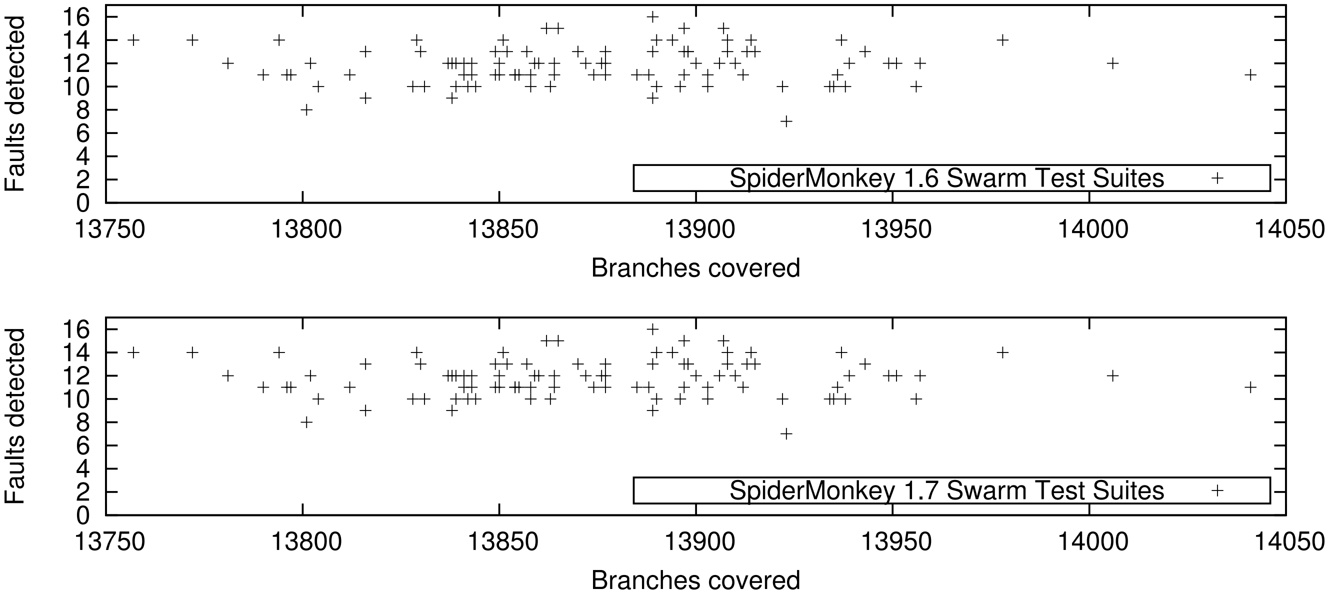

Figure 2. SpiderMonkey 1.6 and 1.7 Swarm Test Suites

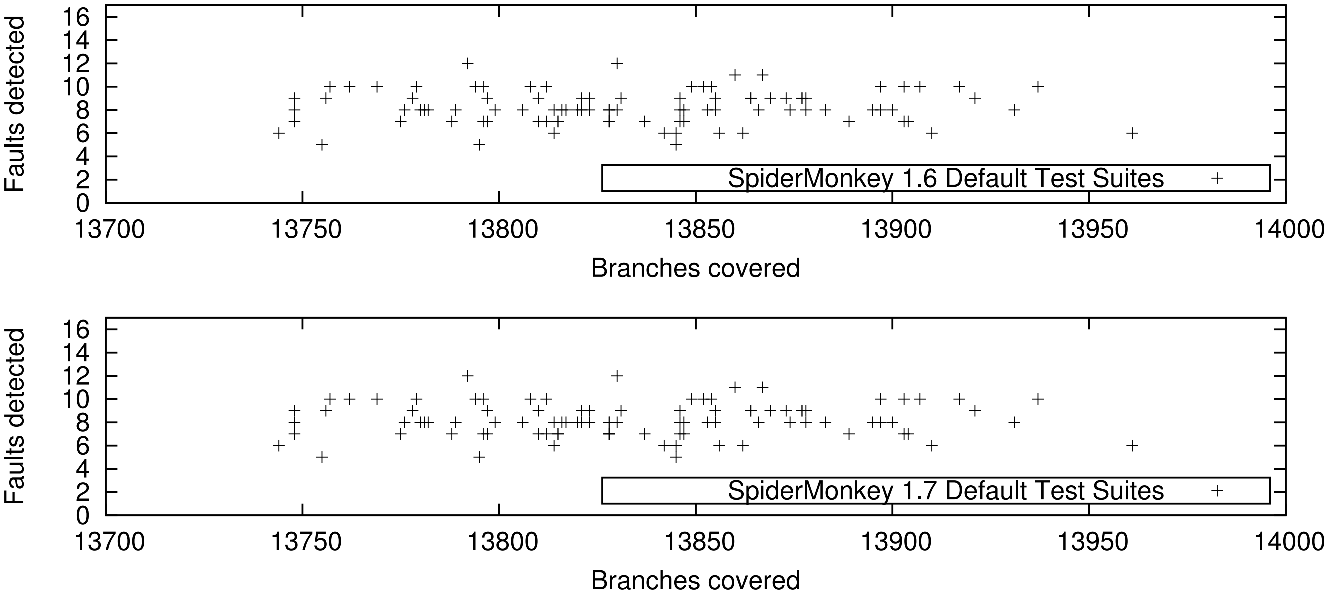

Figure 3. SpiderMonkey 1.6 and 1.7 Default Test Suites

Table 2. τ-b Correlation for Spidermonkey Test Suites, Split by Generation Method
bcov = branch coverage; scov = statement coverage; fcov = function coverage; #t = number of tests; #fail = number of failing tests; #faults = number of detected faults

SpiderMonkey Version 1.6												
	Swarm						Default					
	bcov	scov	fcov	#t	#fail	#faults	bcov	scov	fcov	#t	#fail	#faults
bcov	1.00	0.72	0.40	-0.08	-0.11	0.07	1.00	0.69	0.43	0.18	0.08	0.05
scov	0.72	1.00	0.55	-0.086	-0.21	0.03	0.69	1.00	0.59	0.12	0.05	0.02
fcov	0.40	0.55	1.00	-0.02	-0.09	-0.06	0.43	0.59	1.00	0.10	0.01	-0.03
#t	-0.08	- 0.09	-0.01	1.00	0.16	-0.09	0.18	0.12	0.10	1.00	0.27	0.04
#fail	-0.11	-0.21	-0.09	0.16	1.00	0.09	0.08	0.05	0.01	0.27	1.00	0.11
#faults	0.07	0.03	-0.06	-0.09	0.09	1.00	0.05	0.02	-0.03	0.04	0.11	1.00
SpiderMonkey Version 1.7												
	Swarm						Default					
	bcov	scov	fcov	#t	#fail	#faults	bcov	scov	fcov	#t	#fail	#faults
bcov	1.00	0.60	0.28	0.16	-0.16	-0.07	1.00	0.68	0.34	0.19	0.15	0.22
scov	0.60	1.00	0.47	0.11	-0.01	-0.07	0.68	1.00	0.44	0.08	0.16	0.22
fcov	0.28	0.47	1.00	0.02	0.05	-0.00	0.34	0.44	1.00	-0.03	0.06	0.04
#t	0.16	0.11	0.02	1.00	0.11	0.07	0.19	0.08	-0.03	1.00	0.06	0.21
#fail	-0.16	-0.01	0.05	0.11	1.00	0.22	0.15	0.16	0.06	0.06	1.00	0.22
#faults	-0.07	-0.07	-0.00	0.07	0.22	1.00	0.22	0.22	0.04	0.21	0.22	1.00

erate statistical correlation between the level of coverage a suite achieves and its level of fault detection. This correlation is quite possibly the result of a non-trivial, complex cause that produces both coverage and fault detection, but the existence of such causes is assumed by the use of different methods to produce test suites. Coverage still serves as a useful distinguishing measure for suites, though we should not expect it to always be a significant predictor of fault detection for suites produced by the same underlying method.

The WCH does not propose that in any population of suites coverage will correlate well with fault detection. It says that if suites included have some difference in their method that is not simple randomness, differences in coverage will predict differences in fault detection. Many papers in the literature essentially don't investigate the WCH, because they produce suites by a method where differences are almost exclusively due to randomness (or another simple factor such as size). For example, subsetting a large test suite randomly to produce many smaller test suites [38] is somewhat helpful for investigating the SCH, but is outside the scope of the WCH. It is unlikely that in practice either practitioners or researchers will want to distinguish between suites as homogeneous as random subsets of a single suite. There is no reason to expect much difference in such suites (if they are the same size), and they are not a natural byproduct of debates about testing methods. Studying the WCH makes the experimental data from testing research even more useful, because the utility of coverage in comparing different testing methods is exactly what we want to understand. Because in the real world, two testers seldom have exactly the same level of skill or available resources, it is probably also relevant for comparing human-produced test suites.

6. The Plural of Anecdote is Not Confidence

In the absence of sufficient evidence for the WCH, one presumably safe use for coverage is to examine missing coverage to identify "holes" in a testing effort, as suggested in many books covering testing practice [41, 43] and in some of the experience report literature [26]. However, this use of coverage *also* relies on an assumption: that real missed bugs could often have been identified "if only" test suites had covered certain code, or detected more mutants. How often is this actually true? We don't really know that, either. Even answering the question for a given critical bug can be difficult, without knowledge of the entire history of the system's test effort.

Sometimes it is clear that coverage gaps can serve as signs of potentially missed bugs. For example, the widely publicized SSL vulnerability [42] for Apple's iOS (code shown in Figure 4) is easily identified as soon as test coverage is examined. The bug is the duplication of the `goto fail` statement, where the repetition is outside the scope of the `if` that should guard it. The result is that the value `err` can be returned from the function as 0, despite no actual `sslRawVerify` having taken place to check that the private key included matches the public key for the certificate. Notice that the assignment to `err` calling `sslRawVerify` cannot possibly be executed in any test. This fact should be sufficient to raise alarms in anyone capable of testing this code effectively. On the other hand, the bug also only escapes de-

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
          goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
          goto fail;
          goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
          goto fail;
err = sslRawVerify(ctx,
                   ctx->peerPubKey,
                   dataToSign,                                /* plaintext */
                   dataToSignLen,                        /* plaintext length */
                   signature,
                   signatureLen);
if(err) {
  sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
              "returned %d\n", (int)err);
  goto fail;
 }
fail:
     SSLFreeBuffer(&signedHashes);
     SSLFreeBuffer(&hashCtx);
     return err;
```

Figure 4. Source code for Apple “goto fail” bug

tection from a test suite that *never actually sends an invalid handshake of the right kind* — if a test suite potentially triggers the bug, it probably also checks against the bug. This is something that would be desirable to test in the first place, but is also difficult to test, which may mean that the missed coverage would simply trigger a reaction of “Oh, yes, we don’t check that case, but that’s because it’s a pain to test. We have tests for sslRawVerify itself, of course, which is what matters.” It’s not clear that coverage will actually avoid the bug here, unless there is a “always obtain 100% statement coverage” requirement, which is almost never used in actual practice. Consistent indentation and code review certainly seems like an equally plausible procedure for avoiding “goto fail.” Turning on (and paying attention to) compiler warnings about dead code might also do the trick.

Figure 5 shows an even more infamous bug. As we write, the full consequences of the “Heartbleed” bug [2, 24] are still not known, but it is likely one of the worst security vulnerabilities of all time. As the long and glorious history of buffer access problems in C code might suggest, it turns out that this horrible bug, which may have leaked private key information and leaves no trace of an attack, does not involve subtle flaws in cryptographic algorithms or clever timing attacks. Rather, the problem is a basic C coding issue, where the size of a read in a C memcpy is taken, with complete trust, from a value supplied over the network. The memcpy then happily copies whatever bytes happen to be lying next to the data that is meant to be returned to a buffer and returns this buffer to the adversary on the other side of the network. Data in this area may include passwords, session keys, or even long-term server private keys. For the “goto fail” bug, it was clear that coverage at least theoretically (if not, perhaps, in practice) could help with detection. Could coverage help with Heartbleed?

First, there is certainly no smoking gun as with the Apple bug. The disastrous memcpy and all code surrounding it (including that guarded by conditionals) can be executed as much as desired without exposing the fault. There is likely to be *no difference in branch or statement coverage* between test suites that can expose Heartbleed and test suites that cannot. This isn’t shocking; nobody expects coverage to help detect every fault. We can think of two distinct problems with test suites that (in theory) coverage could expose. The first possibility is that enough code behavior has not been explored — statement and branch coverage are clearly potentially useful for this purpose. Second, a test suite may cause bad behavior in a program, but fail to detect that anything has gone wrong. Weak oracles are a long-standing problem for both conventional and automated testing. One of the strengths of mutation analysis is that, unlike branch or statement coverage, mutation analysis can detect oracle weakness.

Unfortunately, mutation analysis doesn’t appear to help with Heartbleed. In order to detect Heartbleed, a test suite must be able to detect that more bytes have been read from pl than are actually available in it. In theory, tools like Val-

```
int
tls1_process_heartbeat(SSL *s)
        {
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        unsigned short hbtype;
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;

        if (s->msg_callback)
                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

        if (hbtype == TLS1_HB_REQUEST)
                {
                unsigned char *buffer, *bp;
                int r;

                /* Allocate memory for the response, size is 1 bytes
                 * message type, plus 2 bytes payload length, plus
                 * payload, plus padding
                 */
                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
                bp = buffer;

                /* Enter response type, length and copy payload */
                *bp++ = TLS1_HB_RESPONSE;
                s2n(payload, bp);
                memcpy(bp, pl, payload);

                r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Figure 5. Source code for CVE-2014-0160 (Heartbleed)

grind [47] can detect such an invalid read; unfortunately OpenSSL uses a custom allocator that makes the bad read appear innocent. Mutation analysis could (in the unlikely event someone actually manually examines all the mutants that escape detection, which as far as we know is never actually done) expose such weakness, by producing a version of the code that (1) is not equivalent to the original code but (2) can only be seen to differ if the oracle is strong enough to detect a flaw that is detected in the same way as the actual bug. Unfortunately, examining the output of the mutation tool for C code most frequently used in recent work [5], it seems that no mutant of the code actually corresponds to detecting the invalid read. It is possible that *some* mutation operator could force Heartbleed detection, but we do not believe any of the most used conventional operators can do so, and it seems likely that even if a set of mutants large enough to contain one Heartbleed-detecting mutant was generated, no realistic test suite would be likely to kill enough of these mutants to make hand inspection of the un-killed mutants (and thus Heartbleed discovery) practical[8].

7. Advice and Discomfort

To summarize: because it is very difficult for researchers or practitioners to count how many faults a test suite exposes (and for practical applications, the ability to do so would tend to imply there is no need for testing in the first place),

[8] Use of higher-order mutants [39] selected precisely because they are difficult to kill, could be useful here, but bring their own challenges.

we want some easily computed measurement of the quality of a test suite. This measurement should work even for test suites where all tests pass, and should be useful for predicting the suite's ability to actually detect faults. Code coverage is the most commonly used such metric, in part because the idea that at least all statements/branches should be covered if testing is to be effective has intuitive appeal. The use of code coverage as a requirement/goalpost for practical testing efforts or for evaluation of suites in testing research experiments is *dangerous unless code coverage really does correlate (positively) with fault detection*. This is currently an open question, and even recent, more sophisticated, studies of the question do not really show that such a correlation is true. Producing better evidence that coverage really does what we want it to is a difficult problem, in part for the same reasons that we want to use coverage in place of counting faults. Even mutation analysis, which is often assumed to be a good "gold standard" for suite quality measurements, is not really known to be suitable for the purpose. This leads to a state of general discontent on the part of researchers wanting to easily carry out testing experiments and knowledgeable practitioners wanting a practical guide for identifying good test suites. Even the traditional "cautious" uses of coverage (never as a sign of a good suite, but only as a pointer to weaknesses in a suite) lacks truly solid supporting evidence. What is to be done, until more data is available?

7.1 Advice for Researchers

Researchers should probably not evaluate testing techniques only on the basis of coverage, or even only on the basis of mutation analysis. It is hard to quantify the risk that such evaluations are misleading, based on current evidence. The major exception is that using coverage alone should be fine in cases where other work has shown a correlation between coverage and fault detection for the subject program in question. For some benchmarks, this may well hold, though in such cases it isn't clear why the faults from the previous evaluation could not be used. On the other hand, since the evidence that coverage is not usefully correlated with fault detection, when size is held constant, is also weak, adding information on coverage comparisons between suites/techniques as an additional support for a claim of effectiveness could be useful. If the hypotheses about coverage stated above turn out to be true, the papers will benefit from the additional evidence. Certainly if a proposed technique is worse than its alternatives in terms of coverage, this is important to state clearly.

Second, when running testing experiments that will produce test suites with known or estimated fault detection counts, please make use of any available tools to also produce code coverage measures for these suites. While such data may not be useful for the purposes of the research in question (since we all agree that fault detection is the gold standard for evaluation), it can be useful in evaluating hypotheses about coverage. The interested reader should go to `testcoverage.eecs.oregonstate.edu`. This site, which we maintain, will provide information on how to submit information on coverage and fault detection, with sufficient metadata to perform some basic analysis on the coverage hypotheses and analysis of different kinds of coverage. In general, all that is required is information on

1. the software under test (SUT), including source version,
2. the test-generation method used to produce the suite (including all important parameters for tools),
3. at least one measure of coverage (and an identification of the kind of coverage measured),
4. the fault detection capacity of the suite (# bugs),
5. the "size" of the suite as measured by the computation time required to generate and execute that suite.

In practice, there are some further subtle issues, including the particular tool used to measure coverage (a source-code instrumentation based coverage tool's "statement coverage" will not match a byte-code or binary based tool's "statement coverage"), a distinction between generation and execution time for suites (since the suite itself may execute quickly even if massive effort is required to produce the suite), alternative measures of suite size, and, ideally, pointers to source code for SUTs and test generation tools or suites, for reproducibility. Obviously, data with multiple coverage/effectiveness points for the same SUT across multiple generation methods is most useful for WCH purposes, but some insight can be derived even from single suites [22].

7.2 Advice for Practitioners

In some cases where coverage is currently used, there is little real substitute for it; test suite size alone is not a very helpful measure of testing effort, since it is even more easily abused or misunderstood than coverage. Other testing efforts already have ways of determining when to stop that don't rely on coverage (ranging from "we're out of time or money" to "we see clearly diminishing returns in terms of bugs found per dollar spent testing, and predict few residual defects based on past projects"). When coverage levels are required by company or government policy, conscientious testers should strive to produce good suites that, additionally, achieve the required level of coverage rather than aiming very directly at coverage itself [56]. "Testing to the test" by writing a suite that gets "enough" coverage and expecting this to guarantee good *fault detection* is very likely a bad idea — even in the best-case scenario where coverage *is* well correlated with fault detection. Stay tuned to the research community for news on whether coverage can be used more aggressively, with confidence, in the future.

Acknowledgments

The authors would like to thank our anonymous reviewers, as well as Jamie Andrews, Yang Chen, Eric Eide, Michael Ernst, Gerard J. Holzmann, Laura Inozemtseva, Carlos Jensen, Rajeev Joshi, Milos Gligoric, Darko Marinov, John Regehr, Shalini Shamasunder, Rohan Sharma, Willem Visser, and Chaoqiang Zhang for discussions related to this essay. A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786.

References

[1] CodeCover - an open-source glass-box testing tool. `http://codecover.org/`.

[2] Heartbleed bug. `http://heartbleed.com`.

[3] A. Aburas and A. Groce. An improved memetic algorithm with method dependence relations (MAMDR). In *International Conference on Quality Software*, 2014. Accepted for publication.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Trans. Softw. Eng.*, 32:608–624, 2006.

[6] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*, pages 1–10, 2011.

[7] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

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

[9] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *International Symposium on Software Reliability Engineering*, pages 148–157, 1999.

[10] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[11] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[12] F. Del Frate, P. Garg, A. P. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *International Symposium on Software Reliability Engineering*, pages 124–132, 1995.

[13] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Foundations of Software Engineering*, pages 92–104, 2006.

[14] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[15] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Symposium on the Foundations of Software Engineering*, pages 153–162, 1998.

[16] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Trans. Software Eng.*, 19:774–787, 1993.

[17] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology*. Accepted for publication.

[19] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, pages 206–215, 2008. ISBN 978-1-59593-860-2. . URL `http://doi.acm.org/10.1145/1375581.1375607`.

[22] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *International Conference on Software Engineering*, pages 72–82, 2014.

[23] R. Gopinath, C. Jensen, and A. Groce. Mutants: How close are they to real faults? In *International Symposium on Software Reliability Engineering*, 2014. to appear.

[24] M. Green. Attack of the week: OpenSSL Heartbleed. `http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html`.

[25] A. Groce. (Quickly) testing the tester via path coverage. In *Workshop on Dynamic Analysis*, 2009.

[26] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[27] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig, and C. Lopez. Lightweight automated testing with adaptation-based programming. In *IEEE International Symposium on Software Reliability Engineering*, pages 161–170, 2012.

[28] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.

[29] A. Groce, A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *International Conference on Software Testing, Verification and Validation*, 2014.

[30] A. Groce, T. Kulesza, C. Zhang, S. Shamasunder, M. M. Burnett, W.-K. Wong, S. Stumpf, S. Das, A. Shinsel, F. Bice, and K. McIntosh. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Trans. Software Eng.*, 40(3):307–323, 2014.

[31] A. Gupta and P. Jalote. An experimental comparison of the effectiveness of control flow based testing approaches

on seeded faults. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–378, 2006.

[32] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Journal of Software Tools for Technology Transfer*, 10(2):145–160, 2008.

[33] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[34] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[35] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering*, pages 60–71, 2003.

[36] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, 2013. URL `http://philmcminn.staff.shef.ac.uk/publications/pdfs/2013-techreport.pdf`.

[37] M. M. Hassan and J. H. Andrews. Comparing multi-point stride coverage and dataflow coverage. In *International Conference on Software Engineering*, pages 172–181, 2013.

[38] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*, pages 435–445, 2014.

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

[40] 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 *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2014. Accepted for publication.

[41] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001. ISBN 0471081124.

[42] A. Langley. Apple's SSL/TLS bug. `https://www.imperialviolet.org/2014/02/22/applebug.html`.

[43] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670, 9780735619678.

[44] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14: 105–156, 2004.

[45] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 105(33(12)):32–44, 1990.

[46] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *International Symposium on Software Testing and Analysis*, pages 57–68, 2009.

[47] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGSOFT Symposium on Programming Language Design and Implementation*, pages 89–100, 2007.

[48] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 35:1–35:11, 2012.

[49] J. Regehr. Guidelines for research on finding bugs. `http://blog.regehr.org/archives/1038`, 2014.

[50] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.

[51] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability*, 12:219–249, 2007.

[52] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-1789B, RTCA, Inc., 1992.

[53] J. Ruderman. Introducing jsfunfuzz, 2007. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`.

[54] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 262–272, 2005.

[55] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering*, pages 262–277, 2011.

[56] M. Staats, G. Gay, M. W. Whalen, and M. P. E. Heimdahl. On the danger of coverage directed test case generation. In *Fundamental Approaches to Software Engineering*, pages 409–424, 2012.

[57] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2): 203–232, Apr. 2003.

[58] W. Visser, C. Păsăreanu, and R. Pelanek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

[59] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In B. Meyer and M. Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007, pages 194–212. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-25230-3.

[60] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *International Symposium on Software Reliability*, pages 230–238, 1994.

[61] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[62] C. Zhang, A. Groce, and A. Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 60–70, 2014.