AN ABSTRACT OF THE DISSERTATION OF

Mohammad Amin Alipour for the degree of Doctor of Philosophy in Computer Science presented on May 1, 2017.

Title: Leveraging Generated Tests

Abstract approved: _____

Alex David Groce

The main goal of automated test generation is to improve the reliability of a program by exposing faults to developers. To this end, testing should cover the largest possible portion of the program given a test budget (i.e., time and resources) as frequently as possible. Coverage of a program entity in testing increases our confidence in the correctness of that entity.

Generating various tests to cover a program entity is a particularly hard problem to solve for large software systems because the test inputs are complex and they often exhibit sophisticated feature interactions. As a result, current test generation techniques, such as symbolic execution or search-based testing, do not scale well to complex, large-scale systems.

This dissertation presents a test generation technique which aims to increase the frequency of coverage in large, complex software systems. It leverages the information of existing test cases to direct the automated testing. We show the results of the application of this technique to some large systems such as GCC compiler (850K Lines of code), and Mozillas JavaScript engine (120K lines of code). It increases the frequency of coverage upto the factor of 9x, compared to the state-of-the-art technique.

It also proposes non-adequate test-case reduction for reducing the size of test cases by coverage and mutant detection criteria. $C\%$-coverage test reduction technique reduces a test case while preseving at least $C\%$ of coverage in the original test case. N-mutant test reduction technique reduces a test cases while preserving detection of N mutants of the original test case. We evaluate the effectiveness of these test reduction techniques on different attributes of test cases.

This research suggest that the generated test cases should be treated as first-class artifacts in the software development and they can be leveraged for interesting testing tasks.

Leveraging Generated Tests

by

Mohammad Amin Alipour

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 1, 2017
Commencement June 2017

Doctor of Philosophy dissertation of Mohammad Amin Alipour presented on May 1, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Mohammad Amin Alipour, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor Alex Groce. Alex was a true supporter and my champion throughout my doctoral study. Alex allowed me to assume different roles at his research group and collaborate on different projects. His thoughtful feedbacks made my experience at Oregon State University very enriching.

I am grateful to Darko Marinov for many hours that he spent mentoring me on different aspects of graduate school, research, and career. Collaboration with Darko has been one of the best learning experience that I ever had. Thank you, Darko!

Gratitude to my fellow (former) students at the Software Engineering Lab at Oregon Stay university: Ali Aburas, Iftekhar Ahmed, Sogol Balali, Caius Brindescu, Chris Chambers, Souti Chattopadhyay, Mihai Codoban, Rahul Gopinath, Michael Hilton, Rafael Leano, Shane McKee, Nicholas Nelson, David Piorkowski, and Sruti Srinivasa Ragavan for the friendship, insights, and support which made the lab a pleasant place to work.

I owe a great debt of gratitude to my family. I thank my amazing parents, Ebrahim and Saltanat, who their love and emphasis on education motivated me to pursue a career in science. I greatly appreciate my wife, Sahar, and my lovely sons Daniel and Ideen, whom with their support, encouragement, and patience made this journey possible.

CONTRIBUTION OF AUTHORS

Dr. Darko Marinov helped in the inception of the ideas and interperation of the results in the corresponding papers. Agust Shi, Rahul Gopinath, and Arpti Christi contributed in the evaluation and summarization of results.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

There are an increasing number of systems and devices that embody software as their centerpiece. Defects in software systems can cause failures with disastrous effects on the economy, personal life or security of nations. A recent study estimated the software defects impacted 1.1 trillion US dollars in assets, in 2016 [1]. Thus, techniques to assure the correctness of software are highly desirable.

There are three major approaches to increase the confidence in the correctness of existing programs: (1) *dynamic analysis* techniques which are based on observing the behavior of individual runs of software and reasoning about their correctness, (2) *static analysis* techniques which try to reason about (a superset of) all possible executions of a program by examining the source code, and (3) *hybrid dynamic-static* techniques which combine the information of both static and dynamic techniques.

Software testing is a dynamic analysis technique that attempts to increase the confidence in the correctness of programs by observing the behavior of execution programs with multiple inputs and checking them against the expected behavior of the program. Any deviation from the expected behavior can stem from a defect in the program. To expose a software defect in the software under test (SUT), a test case has to reach the buggy statement(s). Then, the execution of the buggy statement should perturb the state of the program to an error state. Finally, the error state that resulted from execution of the buggy statement needs to propagate through the rest of execution of the program to be observed by an external entity or the oracle.

Writing test cases that can meet all the conditions mentioned above is difficult. Moreover, studies [10] have shown that developers do not spend sufficient time writing tests. Thus, automated test case generation is very desirable. Furthermore, as the complexity and size of software systems increase, automated test generation becomes a necessity, because the (in)correctness of complex software systems would have significant economic or social impact.

Automated test generation has been a very active aree of for many years. Over the years, numerous studies proposed novel techniques for generating new test cases (see [4] for a survey of these techniques). However, the generated test cases have been often overlooked. It can be due to the fact that they are being generated by a tool. If a software artifact can be created by a

tool efficiently, developers tend not to store or analyze them.

In this research, our insight is that the generated tests cases can be analyzed or refined for other testing tasks. To this end, we propose: (1) a technique that leverages the statistics in the generated test cases to recommend new configurations for test case generator to direct the testing, and (2) we evaluate two novel test reduction techniques for test cases.

In this research, we focus on the simplest, yet very effective test generation technique: random test cases generators. A random test case generator, as it name suggests, randomly chooses an input from the input space of SUT. Unlike other techniques, it does not have any objective function to maximize or a test target to cover. We demonstrate how the analysis and reduction of seemingly random tests can be profitable in testing.We evaluate our techniques on large C programs, such as the GCC C compiler or the SpiderMonkey JavaScript engine.

1.1 Structure

This dissertation is based on the following two papers.

- *Generating focused random tests using directed swarm testing.* (Chapter 2)
  with Alex Groce, Rahul Gopinath, and Arpit Christi.
  In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016).

- *Evaluating Non-adequate Test-Case Reduction.* (Chapter 3)
  with August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce.
  In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).

Chapter 4 concludes this dissertation and presents potential future work.

1.2 Contributions

Contributions of this research include:

- Introduction of the (to our knowledge) novel goal of increasing the *frequency* with which an automated test generation method produces tests covering specific code targets.

- A novel method (directed swarm testing) for generating *focused random tests*: randomly generated tests that have significantly increased probability of covering selected source code targets.
- Strategies for targeting both individual source code targets and multiple source code targets at once.
- Introduction of two novel test-case reduction approaches: *non-adequate test-case reduction*: $C\%$-coverage and N-mutant reduction.
- Evaluation of the relationship between the size reductions obtained with varying parameters for these reductions, and the code coverage and killed mutants for reduced test cases relative to the original, unreduced test cases.

Generating focused random tests using directed swarm testing

Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi

In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016).
ACM, New York, NY
pp 16-26.

Chapter 2: Generating focused random tests using directed swarm testing

2.1 Introduction

Random testing [32] (sometimes called fuzzing) is now widely recognized as an effective approach for testing software systems, including compilers [52, 53, 60], standard libraries [46], static analysis systems [15], and file systems [30]. Random testing is used in both complex custom-built testing systems (such as those just cited) and simple test harnesses built in a couple of hours. Random testing is often easy to use, widely applicable, and can perform well in theory as well as practice [9]. However, random testing has a few important limitations. One critical limitation is that, for the most part, random testing has little ability (without considerable human effort) to focus on part of a system under test (SUT). Random testers typically target an entire program or module, and have no mechanism for focusing testing on code of particular interest, other than writing a new, customized random test generator.

Much of the efficiency of random testing comes from its blind, undirected nature [64]. It is seldom practical to implement different random testers for all the potential focuses that might be needed, and many powerful random testers [30, 52, 60] tend to be based on generating complete inputs (e.g. programs or function call sequences) as whole system tests; these tools seldom even attempt to provide module-level testing. Of course, tests generated by a random tester can be selected from based on their coverage, but replaying pre-existing tests defeats much of the point of random testing, losing the ability to produce an essentially unlimited number of tests automatically, making effective use of any available testing budget and exploiting massive parallelism.

Techniques for making better use of random tests in situations requiring more focus, such as regression testing, are now appearing [27], but these do not allow the creation of true *focused random tests*: newly generated random tests that are specifically intended to test targeted (for instance, changed) code in a system. Focus can be highly desirable for a variety of reasons. For example, recently changed code is often buggy (perhaps up to one third of code changes introduce some bug [39]). Moreover, newly changed code has, by definition, been far less tested than long-standing code, especially in systems where aggressive random testing is applied routinely.

At present random testing does not even support an easy way to direct testing to aggressively cover changed code. In addition to changed code, focused random tests are useful in any cases where a part of a system is suspected to be more fault-prone or difficult to cover than the remainder of the SUT. The inability to perform efficient targeted testing is a real deficiency in random testing.

While some other techniques (symbolic execution [22, 59] and search-based techniques [34, 44]) for test generation allow for targeting of specific source code, those techniques usually have not been scaled to the generation of, e.g., whole-program inputs for industrial strength compilers[1]. Hand-tooled whole-program random testers, however, are a popular technique for testing such systems, including C compilers [41, 60], JavaScript engines [36, 52, 53], and Google's Go language [56]. More critically, search-based and symbolic techniques are designed to support the generation of *a test* that covers a desired target, not the production of an arbitrary number of different tests hitting a target. For example, most search-based systems attempt to produce *one* test for each coverage target, and consider a statement tested once it has been covered once, only covering it additionally as needed to cover other targets. While very useful for generating a high-coverage suite, this does not address the need to test a suspect statement in a diverse and essentially unlimited number of ways, given sufficient compute resources. Focused random tests combine the nearly unlimited novelty of random test generation with the ability to target testing to code of particular interest, without forcing developers to write custom random testers for code components.

In this paper, we propose a method, *directed swarm testing*, that makes the generation of random tests that focus on selected code targets possible. Using swarm testing [31] (a variation of random testing) and recording statistical data on past testing results [25] enables generation of new random tests that target (that is, have higher probability of covering, and thus higher coverage frequency for) any given source code element, usually without modifying an existing, highly-tuned random tester. This ability has further uses than just simple change-based "regression testing"; for example, a compiler developer using Csmith [60] and concerned about the correctness of a particular set of seldom-executed lines in a complex optimization's implementation may apply this technique. Assuming that data on past testing has already been collected, the process can be as simple as putting the source lines of interest into a file and running a simple

[1] SAGE [21] for symbolic execution, and the work of Kifetew et al. [38] for search-based testing are promising exceptions, though in the first case only a limited evaluation over coverage, not faults, was performed, and in the second case the Rhino JavaScript tests are arguably more limited than those generated by `jsfunfuzz`.

script that launches in parallel a large set of Csmith instances tuned to have high coverage of the suspect code. In our experiments, the fraction of tests that cover targeted code was improved by up to nearly 9x over running the random tester as usual, and the improvement is typically on the order of 2x or more. The more rarely code is covered in undirected tests — so long as it has been covered enough in past data to make a basis for statistical analysis — the more its coverage frequency can be boosted.

To our knowledge, this goal of increasing frequency of coverage (as opposed to generating at least one test hitting a coverage target, a common goal of testing methods) is both novel and clearly useful. The goal is, in a sense, incomparable to the goal of covering never-before-covered code targets, since our assumption is that some test(s) hitting the targeted code already exist; we aim to produce *many more* tests hitting the targets, since it is well known that for most faults it is not sufficient simply to cover the faulty code — it must be covered under the right conditions. This motivates producing a diverse set of tests covering any code warranting extra attention, whether that code is suspicious due to modification, static analysis warnings (that may be false positives), code smells, or any other heuristics for potential faults.

Our experimental results show that, for single targets, across *all* strategies proposed, directed swarm testing improves the fraction of tests that hit a target by 3.5x on average for YAFFS2, 2.5x on average for GCC, and 1.6x on average for SpiderMonkey. Directed swarm testing improved coverage for 100%, 95%, and 69.5% of targets (again, across all strategies) for YAFFS2, GCC, and SpiderMonkey respectively. Results for multiple targets are more complex, but still promising, though as the number of targets increases the effectiveness over swarm testing decreases (as it must, in the limit: targeting all code is equivalent to targeting none). We compare our method both against the baseline random test generators (hand-tooled optimized random testing) and modified test generators using swarm testing.

Contributions of this paper include:

- Introduction of the (to our knowledge) novel goal of increasing the *frequency* with which an automated test generation method produces tests covering specific code targets.
- A novel method (directed swarm testing) for generating *focused random tests*: randomly generated tests that have significantly increased probability of covering selected source code targets (Section 2.3).
- Strategies for targeting both individual source code targets and multiple source code targets at once (Section 2.4).

- Empirical results showing the effectiveness of these strategies on large real-world software systems and test generators with complex test features (the YAFFS2 flash file system, the GCC compiler, and Mozilla's SpiderMonkey JavaScript engine) (Sections 2.5 and 2.6).
- Empirical results of effectiveness of these strategies on finding *real faults* in a large software system (Sections 2.5 and 2.6).

2.2 Preliminary Concepts

Swarm testing [31] is a testing approach that improves the diversity of tests by randomizing the configuration of a test generation system (typically a random tester, though it is also applicable to model checking [2, 26]). The idea behind swarm testing is simple: most random test generators support a natural concept of *features*. A feature is a property of a test case that can be controlled by a test generator. A configuration of a test generator is often defined by a set of features. For example, in grammar-based testing, features are usually terminals or productions in the grammar, and in API-based testing each function or method call is a feature. The traditional approach to random testing is to always make all features available in the construction of each test. Swarm testing, in contrast, randomly chooses (with base probability of 50%) which features to include in each test, omitting about half of all available features in each test. This often increases the effectiveness of testing due to interactions between features, and the fact that, since tests are limited in size, including many features necessarily means including less of each individual feature. Swarm testing has been recognized as essential to getting good results from compiler fuzzers [41] and has sometimes nearly *doubled* fault detection and/or coverage for mature random testers [25]. Swarm testing has also been applied to the CCG C compiler testing tool [45] and the GoSmith [56] fuzzer for Google's Go language, and a Constraint Logic Programming technique extending the ideas in swarm testing has been used to discover faults in the Rust type system [17].

Adapting most random testers to support swarm testing is simple. Features are often opportunistically chosen to match existing configuration. For example, Csmith supports numerous controls on C code generated, in order to, e.g., test compilers with known bugs. Using Csmith with a configuration simply requires calling it with command line arguments (e.g., `csmith --no-pointers --no-structs --no-unions`). For `jsfunfuzz` configuration of features for generating JavaScript code was introduced using a 50-line Python script that con-

```
static uint16_t  func_1(void) {
    uint16_t l_24[3][2] = {{0xD44FL, 0xD44FL},
      {0xD44FL, 0xD44FL}, {0xD44FL, 0xD44FL}};
    return l_24[1][1]; }
int main (int argc, char* argv[]) {
    func_1();
    return 0; }
```

(A) Simplified random test case generated by Csmith, (boilerplate removed). This test case features arrays but does not feature pointers, structs, jumps, or volatiles.

```
tryItOut("L: {constructor = __parent__; }");
tryItOut("prototype = constructor;");
tryItOut("__proto__ = prototype;");
tryItOut("with({}){__proto__.__proto__=__parent__;}");
```

(B) Simplified random test case (without `jsfunfuzz` infrastructure) for SpiderMonkey JavaScript engine. Features here include labels, assignments, and `with` blocks, but do not include `try` blocks, infinite loops, or XML.

Figure 2.1: Features for Random Test Cases

siders each choice in the recursive code generator to be a feature. Random testing based on API calls is usually trivial to modify to exclude calls at will, as in our YAFFS2 tester. Figure 2.1 shows examples of features for C and JavaScript tests. Note that a feature can be a relatively simple grammatical construct or, depending on how tests are generated, a more complex semantic feature (e.g., irreducible control flow). Given a configuration, a tester can usually generate an unbounded number of different tests containing (at most) those features.

2.2.1 Triggers and Suppressors

A *target* is any behavior of the SUT that is produced by some (but usually not all) test cases. The most obvious targets are faults and coverage entities, e.g.: whether a test case exposes a given fault, whether a given block or statement is executed, whether a branch is taken, or whether a particular path is followed. Hence, faults, blocks, branches, and paths are targets and a test case *hits* a target if it exposes or covers it. Given the concepts of features and targets, we can ask whether a feature f "helps" us to hit a target t: that is, are test cases with f more likely to hit t? That some features are helpful for some targets is obvious: e.g., executing the first line of a method in an API library usually *requires* the call to be in the test case. Less obviously, features may make it *harder* to hit some targets. For example, finite-length tests of a bounded stack that contain `pop` calls are less likely to execute code that handles the case where the stack is full, closing files may make it harder to cover complex behavior in a file system, and including

pointers in a C program prevents some optimization passes from running [31].

There are three basic *roles* that a feature f can serve with respect to a target t: a *trigger*'s presence makes t easier to hit, a *suppressor*'s presence makes t harder to hit, and an irrelevant feature does not affect the probability of hitting t. The relation between features and targets can be non-trivial to predict and understand in large programs with complex features.

In previous work [25], it was shown that for all non-trivial SUTs examined, most targets had a few triggers and a few suppressors. We adopt from that work a formal definition of trigger and suppressor features based on Wilson scores [58] over hitting fractions in pure (undirected) swarm testing. Given feature f, target t, and test case population P where f appears in tests at rate r, compute a Wilson score interval for a given confidence (e.g., 95%) (l, h) on the proportion of tests hitting t that contain f. If $h < r$, we can be, e.g., 95% confident that f suppresses t. The lower h is, the more suppressing f is for t. When $l > r$, f is a trigger for t. If neither of these cases holds, we can say that f is irrelevant to t. The appropriate bound (lower or upper) may then be used as a conservative estimate for the true fraction F of tests with f hitting t:

$$F(f,t) = \begin{cases} r & \text{iff } l \leq r \leq h; \quad \text{(irrelevant)} \\ l & \text{iff } l > r; \quad \text{(trigger)} \\ h & \text{iff } h < r. \quad \text{(suppressor)} \end{cases}$$

F is easily interpreted when the rates for features are set at 50% in P, as in normal swarm testing. Critically, because of the way swarm testing works, feature/target relationships are always causal, evidence of a genuine semantic property of the SUT [25].

2.3 Directed Swarm Testing

We can exploit the fact that most targets of real-world SUTs have both triggers and suppressors to focus swarm testing on a given target, or set of targets. *Directed swarm testing* is performed similarly to conventional swarm testing, and like swarm testing, usually requires little or no modification of the base test generator. The difference between directed swarm testing and conventional swarm testing is that, instead of using completely random configurations, directed swarm testing uses configurations *based on the trigger and suppressor information collected for a single target or a set of targets*. Rather than a single algorithm, directed swarm testing is a family of strategies for choosing features in testing, with one constraint: when targeting t,

directed swarm testing never uses configurations containing any suppressors of t.

When directed swarm testing is applied to multiple targets $\mathcal{T}$ at once, as is often useful in testing changed code, it may only target some subset of $\mathcal{T}$ in each individual test generation. A directed swarm testing strategy is effective if it increases the average rate at which tests hitting targets t are generated above the base rate for non-directed swarm testing. The larger the increase, the more effective the directed swarm testing strategy.

A typical application of directed swarm testing could be targeting changes made to the SUT. A developer has just implemented a new feature, and in the process added a new function `f` to the code, modified four lines of code in an existing function `g`, and added calls to `f` in three locations scattered throughout the program, all guarded by an existing conditional. The developer can run existing regression tests [27], and run an existing random tester in swarm mode, to detect bugs in the new feature. However, the function `g` is called by relatively few regression tests, and undirected swarm testing only calls `g` once in every twenty tests. The calls to `f` are only slightly more frequent. Assuming the unmodified code is correct, many of the tests generated in undirected swarm testing will be useless. Fortunately, it is easy to construct a set of targets for directed swarm testing in this situation: the modified lines in `g` are obvious targets, and previous random testing results should contain enough information to calculate their triggers and suppressors with high confidence. The code for `f`, in contrast, is new; the developer has no information on triggers and suppressors for `f` itself. However, the developer always has information on some existing code that precedes new code to be targeted, and is as close as possible to it in the revised CFG for the SUT (the proof is trivial: if new code has no such nodes, it is either unreachable in the CFG, or the new code is the first node in the CFG, in which case it is always called and does not need to be targeted)[2]. The developer performs directed swarm testing, using this set of targets, and, if directed swarm testing is effective, is able to either find a bug or establish that the new code is likely correct much more quickly, since she has increased the frequency with which tests validate the changes. The measure of success is *how many* tests covering changed code are produced within a given testing budget (or how quickly a fault is detected, when the code is faulty).

A major advantage of directed swarm testing is that, like swarm testing, it has essentially the same extremely low overhead as all random testing. The only additional cost for directed swarm testing is to collect coverage information when running some swarm tests, in order to

[2]It might also be possible to use code dominated by the changed code as a target, but there does not always exist any such code.

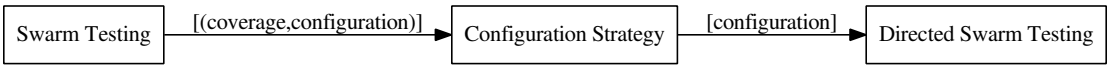

Figure 2.2: Workflow of directed swarm testing.

compute triggers and successors for a program. Running some random tests with coverage instrumentation is already a common practice in aggressive testing, so this is hardly a major burden, even with the need to re-baseline trigger/suppressor information as code evolves over time. In previous work, triggers and suppressors for lines of code that continued to exist through many software versions did not change dramatically, even from major release to major release, for Mozilla's SpiderMonkey JavaScript engine [25]. In short: baselining is cheap, part of existing good testing practice, and there is considerable evidence that re-baselining of coverage relationships can be performed infrequently in various testing applications [27, 28, 54].

2.4 Configuration Strategies

Figure 2.2 shows the overall workflow of directed swarm testing, which is simple. First, swarm testing is performed as usual, without any targets, to detect faults and collect coverage information over the entire SUT. In order to apply directed swarm testing, the only information from this testing that is required is the set of (coverage, configuration) tuples for all tests generated in undirected swarm testing. This information can, as described in the introduction (Section 2.2.1) and in more detail in the empirical work of Groce et al., [25], be used to compute, for each source code target t (in this paper's experiments, a statement), the set of triggering features $T(t)$, suppressing features $S(t)$, and irrelevant features $I(t)$. The heart of a directed swarm testing method is a strategy for producing configurations for new tests based on $T(t)$, $S(t)$, and $I(t)$. This can be done for a single t or for a set of targets $\mathcal{T}$. While the idea that knowledge of triggers and suppressors should enable us to improve testing for targets seems clear, there are trade-offs to consider in determining the actual configurations to use in testing for targets. Most importantly, the triggers and suppressors are determined with respect to a distribution of test cases such that most tests have about half of all features enabled; causal patterns may change when using a very different configuration distribution. While hitting the targets is important, it is also essential to maintain some test diversity to maximize the value of each individual test run — after all, simply running a single chosen test case that hits a target (with mutation fuzzing) may "maximize"

target coverage, but loses almost all advantages of random testing.

2.4.1 Single-Target Strategies

We first consider the simplest case, targeting a single source code element. This is likely to be a very common goal, even for regression testing. If a developer only changes code in a single basic block, it is essentially one target with one set of triggers and suppressors (since the coverage vectors for all statements in a basic block are necessarily the same). Even modifying a few lines of code that are nearby in the CFG of the SUT is probably likely to involve similar triggers and suppressors, in most cases. In fact, multiple nearby targets can probably be effectively targeted in most cases by choosing their nearest common control flow dominator (for example, when all the modified code is in a single function)[3].

We propose three basic strategies for a single target, t:

1. **Half-swarm:** The Half-swarm strategy produces configurations for testing in the same way as undirected swarm testing, with the exception that all features in $S(t)$ (the suppressors) are omitted from each configuration and all features in $T(t)$ are included in each configuration. It can be trivially implemented by applying an AND mask for suppressors (with all 1 bits except for suppressors) and an OR mask for triggers (with all 0 bits except for triggers) as a final stage in undirected swarm testing. In other words, a configuration $C_i = \{f | f \in T(t) \cup randomSample(f | f \notin S(t))\}$, where $randomSample$ returns a random sample of a set such that each element has a 50% chance of being included.

2. **No-suppressors:** The No-suppressors strategy uses only one configuration, which includes all triggers and irrelevant features, but no suppressors: $C = \{f | f \notin S(t)\}$.

3. **Triggers-only:** The Triggers-only strategy, as the name suggests, also uses a single configuration for all testing, where all triggers are included and no other features are included: $C = \{f | f \in T(t)\}$.

The motivation for **Half-swarm** is that swarm testing is effective, and directed swarm testing should, perhaps, remain as close to undirected swarm testing as possible, except for taking triggers and suppressors into account. The motivation for the other two strategies is that while

[3] A common statement dominated by all targets can also be used, if such a statement exists.

swarm testing is effective for general testing of an SUT, it may not be ideal when generating focused random tests. The diversity that makes swarm testing useful may be useless or harmful for increasing frequency of coverage for a single target; however, it is not clear if a minimal or maximal configuration that respects triggers and suppressors would be best, given this assumption. **Triggers-only** uses a minimal configuration, with only those features known to improve coverage of the target included, while **No-suppressors** is maximal, only omitting features known to hinder coverage of the target. The computational cost for all techniques is the same (and essentially identical to that of non-directed swarm testing or pure random testing). As we see below, in addition to the basic empirical question of effectiveness, the idiosyncracies of some random testers may also determine which of these strategies should be chosen. In particular, for some testers, if very few features are present in a configuration, it may not generate any valid tests. When there are many features and a 50% chance of inclusion, the problem does not arise, but using Triggers-only may frequently fail to generate valid configurations.

2.4.2 Multiple-Target Strategies

For multiple targets, $\mathcal{T}$, our strategies reduce the problem to that for single targets $t \in \mathcal{T}$:

1. **Round-robin:** The Round-robin strategy simply applies a single-target strategy in a round-robin fashion, for $t \in \mathcal{T}$.

2. **Merging:** The Merging strategy attempts to *merge* triggers and suppressors for targets in $\mathcal{T}$ to produce a minimal set of meta-targets, then uses round-robin.

The motivation behind **Round-robin** is simple: to cover a set of targets, split the testing time between those targets. If multiple targets have similar suppressors and triggers, we may end up covering a target with tests not aimed at that target, but the basic idea is simply to assume all targets are equally important and cannot be tested at once. Round-robin is parameterized on a single-target strategy.

Merging approaches are more complex. They are motivated by an observation: if for two targets, t_1 and t_2, $\neg\exists f.(f \in S(t_1) \wedge f \in T(t_2)) \vee (f \in T(t_1) \wedge f \in S(t_2))$, then there may be no reason we have to target t_1 and t_2 with different configurations. They do not have any *conflicts*, where a conflict is a feature that suppresses one target but triggers the other target.

Algorithm 1 illustrates one simple algorithm to produce a set of targets $\mathcal{T}'$ for targets $\mathcal{T}$. Given targets $t_i, t_j \in \mathcal{T}$, we say t_j *subsumes* t_i, denoted $t_j \sqsupset t_i$, if and only if, $S(t_i) \subset$

$S(t_j) \wedge T(t_i) \subset T(t_j)$. In other words, t_j requires a *stricter* combination of features than t_i. **Subsumption** merging removes t_i and only keeps the stricter combination of features, assuming that it will test both targets. The computational cost of the algorithm is quadratic in the number of targets to consider merging (and thus negligible for likely sets of targets).

Algorithm 1 Algorithm for Merging using Subsumption only.

for $\forall t_i \in \mathcal{T}$ **do**
  if $\exists t_j \in \mathcal{T} | t_j \sqsupset t_i$ **then**
    $\mathcal{D} = \mathcal{D} \cup t_i$
  end if
end for
`return` $t \in \mathcal{T} | t \notin \mathcal{D}$

Algorithm 2 Algorithm for Aggressive Merging, with randomized approximation of optimal merges (n = # of trials).

$\mathcal{B} = \mathcal{T}$
for $i = 0 \ldots n-1$ **do**
  $\mathcal{M} = \mathcal{T}$
  while $\exists t_i, t_j \in \mathcal{M} : t_i \neq t_j \wedge (\neg \exists f.(f \in S(t_i) \wedge f \in T(t_j)) \vee (f \in T(t_i) \wedge f \in S(t_j)))$ **do**
    `pick` t_i, t_j
    $T(t_m) = f | f \in T(t_i) \vee f \in T(t_j)$
    $S(t_m) = f | f \in S(t_i) \vee f \in S(t_j)$
    $I(t_m) = f | f \notin T(t_m) \wedge f \notin S(t_j)$
    $\mathcal{M} = \mathcal{M} \cup t_m - t_i - t_j$
  end while
  if $|\mathcal{M}| < |\mathcal{B}|$ **then**
    $\mathcal{B} = \mathcal{M}$
  end if
end for
`return` $\mathcal{B}$

It is also possible to merge in a more **Aggressive** fashion. In the absence of conflicts, we can in principle merge *any* two targets even where neither is stricter than the other, treating them as one target t', with $T(t') = f | f \in T(t_1) \vee f \in T(t_2)$, $S(t') = f | f \in S(t_1) \vee f \in S(t_2)$, and $I(t') = f | f \notin S(t') \wedge f \notin T(t')$. In this way, we can keep merging targets (replacing the two non-conflicting targets with the new meta-target) without conflicts to produce a small

set of configurations that are directed at many targets at once. However, finding the merges to produce a truly minimal set of configurations is in NP-complete, equivalent to the optimal tuple merge problem [50]. We implemented an SMT-based exact solver for merging targets using Z3 [16], which was able to construct perfect solutions for up to 20 targets (typically solving for 300 features in less than 2 minutes, but sometimes taking more than 10 minutes), but did not scale to 40 targets at all, even with very few features (timing out after many hours). Fortunately, due to the fact that most targets have either absolutely few (< 3) triggers and suppressors or at least relatively few ($< 5\%$ of features) triggers and suppressors [25], random ordering of matches (using the best solution after a fixed number of trials) approximates exact solutions effectively and quickly. In our experiments, a random approximation of optimal merging, even using 1,000 trials, always produced a nearly-optimal set of configurations (at most one larger than the optimal set produced by Z3) in less than 1 second, for up to 20 targets. In experiments, we used 10,000 trials. Algorithm 2 shows the randomized algorithm for Aggressive Merging of targets. We assume that Subsumption Merging has already been applied before this algorithm is called.

Both the Subsumption and Aggressive Merging strategies are, like the Round-robin strategy, parameterized on a single-target configuration strategy. It is, in part for this reason, not clear whether (and how much) we *should* merge configurations. Merging targets produces "more specialized" configurations that leave little room for the basic single-target strategies to operate (because merging increases the numbers of fixed triggers and suppressors for each merged target). Round-robin maintains maximal configuration diversity (consistent with directing the testing). Subsumption Merging assumes that when one target subsumes another, they are truly similar and can be tested in the same way. Aggressive Merging uses as few configurations as possible, but may result in a very small number of targets with very few irrelevant features. Whether such targets can actually be effectively tested by the same configurations is not obvious without empirical investigation.

2.5 Evaluation Methodology

We used three medium-moderately large C programs (shown in Table 2.1) to evaluate directed swarm testing. While not extremely large, these are all important systems-software programs, the typical of the kind of program for which an effective dedicated random tester can be expected to exist. For YAFFS2 (formerly the default image file system for Android), we used custom

Table 2.1: Experimental Subjects

SUT	LOC	Fuzzer	Description
YAFFS2	15K	`yaffs2tester`	Flash File System
GCC 4.4.7	860K	Csmith	C and C++ Compiler
SpiderMonkey 1.6	118K	`jsfunfuzz`	JavaScript Engine For Mozilla

test generation tools descended from those used to test the file systems for NASA's Curiosity Mars Rover [30], and applied in previous work on combining random testing and symbolic execution [64]. For GCC, we used the Csmith [60] C compiler fuzzer to generate tests. Csmith is a highly effective tool that has been used to detect more than 400 previously unknown bugs in GCC, LLVM, and other production C compilers. For SpiderMonkey, Mozilla's JavaScript engine, we used `jsfunfuzz` [52], a well-known JavaScript fuzzer responsible for finding more than 6,400 bugs in SpiderMonkey, combined with a small Python script to add swarm testing. The other two test generators already supported swarm testing.

Our subjects were chosen with two criteria in mind: first, they represent different kinds of features for swarm testing. YAFFS2 features are API calls, but (unlike the Java libraries more commonly used in the literature of API-call test generation), the calls modify a single, very complex program state (the file system itself) with complex dependencies. Features for SpiderMonkey testing using `jsfunfuzz` are actual production rules in a recursive generator, very difficult for a human engineer to understand (but easy to implement in a swarm tester). The complex recursive generation makes it an interesting subject to gauge the limits of our technique. Finally, test features in Csmith [60] are high-level semantic features of C programs, some of which do not correspond to simple grammar productions, and the features were devised to help compiler engineers deal with compilers with limited support for various C features, not for use in swarm testing. Second, we wanted our subjects to be representative of the kinds of system software subjected to aggressive, sophisticated random testing.

Table 2.2 shows parameters for our experiments. In this table: # Features shows the number of features in the SUT that can be tested by the corresponding fuzzer, seed time shows time spent in minutes to generate the initial (undirected swarm) test suite that is used for extracting trigger/suppressor features for statements, and directed time shows the time spent for directed testing of targets. The stochastic nature of random testing required us to run experiments multiple times to ensure results are statistically significant. For each test subject we generated between 30 and 60 initial test suites (# Suites) using undirected swarm testing. We collected data on

Table 2.2: Experimental Parameters.

SUT	# Features	Seed time (min.)	Directed time (min.)	# Undirected Suites
YAFFS2	43	15	5	60
GCC	25	60	10	30
SpiderMonkey	171	30	10	54

configurations and coverage from these tests, and computed Wilson scores (and thus triggers and suppressors) for all statements covered in the tests. For each such test suite, we picked up to 35 sets of random targets (statements), with sizes 1, 5, 10 and 20 (up to 5 for each size) to evaluate directed swarm testing[4]. We also used the default configuration of each test generator to produce one traditional (non-swarm) random test suite for each swarm test suite produced (thus from 30-60 pure random suites), to compare the effectiveness of directed swarm testing and traditional random testing, using the same time budget.

We randomly chose targets (statements) covered by 10% to 30% of test cases in the original test suite, to restrict evaluation to targets that are at least somewhat difficult to cover, but for which a statistical basis for directed swarm testing definitely exists. For more rarely covered targets, where triggers and suppressors are less certain, the nearest control-flow dominator with sufficient coverage in tests can be used as a replacement target. Note that with a large amount of historical coverage data, as might be collected in an overnight test run on a stable version, many more targets would have statistical support for accurate triggers and suppressors. The 10%-30% selection is only to enable experiments using limited coverage data, not a limitation of directed swarm testing.

For the single-target sets we applied each of the Half-swarm, Triggers-only, and No-suppressors strategies. For all multiple target sets, we also applied Round-robin, Subsumption, and Aggressive strategies (in each case paired with a single-target strategy, for nine strategies in all). We varied the time for undirected testing and directed testing according to suite complexity in each case. In total, we ran tests for slightly more than 3,000 hours and generated over 20,000 test suites.

[4]We also collected data for size 2, 3, and 4 target sets, which will be provided in a technical report; in the interests of space, these results, which shed little light on multi-target strategies and were similar to results for size 5, are omitted from this version.

Our primary measure of effectiveness is simple. For any test suite, we compute the hitting fraction HF for tests that cover a target t (if there are n tests in a suite and m tests cover t then,$HF = \frac{m}{n}$) — if every test in a suite covers t, $HF = 1.0$ and if no tests cover t, $HF = 0.0$. Suppose the hitting ratios in an undirected suite and directed suite are HF_u and HF_d respectively, we use the ratio $\frac{HF_d}{HF_u}$ to measure the effectiveness of directed testing in hitting targets more frequently. Note that directed test suites with $\frac{HF_d}{HF_u} > 1.0$ offer improvement over undirected test suites. This is the measure a developer wants to increase via targeting.

2.6 Results

Our experimental results address six basic research questions:

- **RQ1:** (How much) does directed swarm testing improve coverage for single targets?
- **RQ2:** Which strategies for single-target directed swarm testing are most effective?
- **RQ3:** (How much) does directed swarm testing improve coverage for multiple targets at once?
- **RQ4:** Which strategies for multiple-target directed swarm testing are most effective?
- **RQ5:** Can directed swarm testing help detect actual faults?
- **RQ6:** How much does directed swarm testing improve coverage over traditional random testing?

Figure 2.3 illustrates the distribution of targets' hitting fraction (HF) for (undirected) swarm testing and directed swarm testing. It shows that, in most cases, the hitting fraction for targets in directed swarm testing is much higher than the hitting fraction for undirected swarm testing. For brevity, in the rest of this section, we use "directed swarm testing" and "directed testing" interchangeably, as directed swarm testing is the only directed testing approach we evaluate (and, to our knowledge, the only one applicable to our subject programs).

Table 2.3 provides much more detailed information about the performance of directed testing for single-target directed testing[5]. It summarizes $\frac{HF_d}{HF_u}$ for different strategies. In this table,

[5]Tables containing detailed results for multi-target testing were omitted due to space limitations, and appear in tech report online.

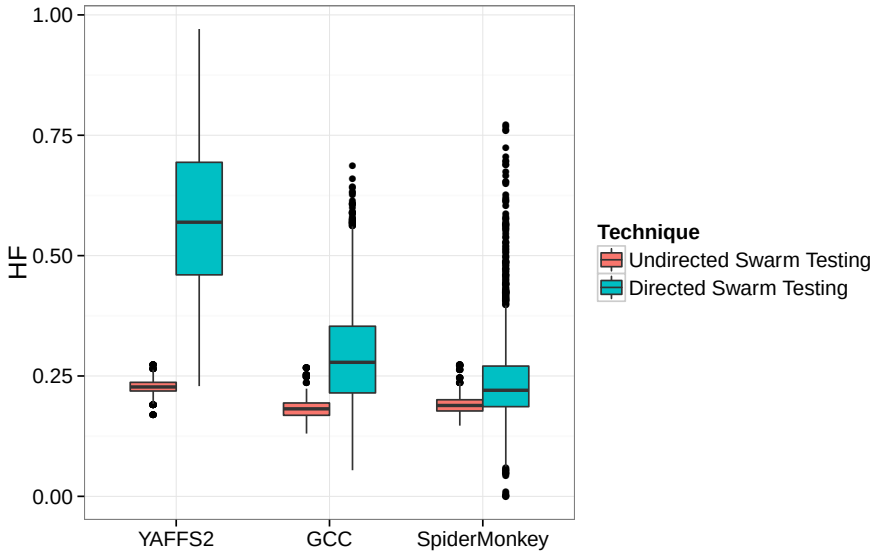

Figure 2.3: Hitting fraction in undirected swarm testing (HF_u) versus directed swarm testing (HF_d) over all strategies.

"count" contains the number of test suites generated by directed swarm testing using strategies described in the corresponding row. The $\frac{HF_d}{HF_u} > 1.0$ column shows the fraction of test suites where target(s) were covered more often by the directed swarm testing than the corresponding initial undirected swarm testing. For example, the value 0.8 in this column means: in 80% of test suites generated by directed swarm testing, the HF for targets is higher than the original undirected swarm. "mean","std. dev", "min", "25%", "50%", "75%" and, "max" respectively denote average, standard deviation, minimum, first quartile, second quartile (i.e. median), third quartile and maximum of $\frac{HF_d}{HF_u}$ in test suites generated by corresponding strategies in each row.

2.6.1 RQ1 and RQ2: Single-Target Strategies

Table 2.3 shows the results for single-target directed swarm testing under different directed testing strategies, including p-values for Wilcoxon tests. Figure 2.4 visualizes these results. Table 2.3 shows that directed swarm testing has been successful in increasing HF for targets for YAFFS and GCC. For all strategies with YAFFS, directed swarm testing always increased hit-

Table 2.3: Results for single-target directed random testing.

Strategy	$\frac{HF_d}{HF_u} > 1$	count	mean	std. dev	min	25%	50%	75%	max	p-val
YAFFS2										
Half-swarm	1.0	218.0	3.56	0.59	1.38	3.11	3.7	3.96	5.01	0.0
No-suppressors	1.0	216.0	3.03	0.7	1.03	2.4	3.19	3.56	4.44	0.0
Triggers-only	1.0	218.0	3.94	0.64	2.26	3.57	4.0	4.25	7.87	0.0
GCC										
Half-swarm	0.99	138.0	2.4	0.99	0.94	1.69	2.19	3.0	6.33	0.0
No-suppressors	0.94	135.0	2.56	1.0	0.0	1.86	2.59	3.23	5.58	0.0
Triggers-only	0.92	129.0	2.28	1.0	0.53	1.53	2.13	2.94	5.29	0.0
SpiderMonkey										
Half-swarm	0.73	260.0	1.75	1.06	0.0	0.88	1.74	2.49	4.39	0.0
No-suppressors	0.65	260.0	1.15	0.62	0.0	0.61	1.27	1.6	3.14	0.30234
Triggers-only	0.84	19.0	4.56	3.01	0.11	2.6	3.62	7.23	8.82	0.0006

ting ratio. The hitting fraction of targets using directed swarm testing was more than three times more than the hitting fraction in the undirected testing, on average. For GCC, directed swarm testing increased the hitting fraction of targets for more than 90% of targets. On average, the directed testing increased the hitting fraction of targets by a factor of 2 or more.

The results for SpiderMonkey are mixed partly because the design of `jsfunfuzz` is such that, if we remove certain features, the fuzzer cannot produce any test cases at all. Moreover `jsfunfuzz` encodes SpiderMonkey's feature set by paths through a complex recursive code generation system that resembles a grammar. In many cases, with SpiderMonkey, the triggers for a target are low-level productions that are only reachable through top-level parts of the fuzzer that correspond to irrelevant features — they are highly redundant. This makes it hard to identify triggers and suppressors, since the chance of undirected swarm generating a configuration disabling all paths is small. However, even for SpiderMonkey, directed swarm testing increases the hitting fraction of more than half of targets, and Half-swarm had mean improvement close to 2x. Note that most configurations for the Triggers-only strategy could not generate any test cases.

Observation 1: Directed swarm testing, with the exception of one strategy for SpiderMonkey, significantly ($p < 0.01$) increases coverage frequency over undirected testing.

The average improvement for single-target directed testing ranges from 1.15x for SpiderMonkey with the No-Suppressor strategy to nearly 4x for Triggers-only with YAFFS2.

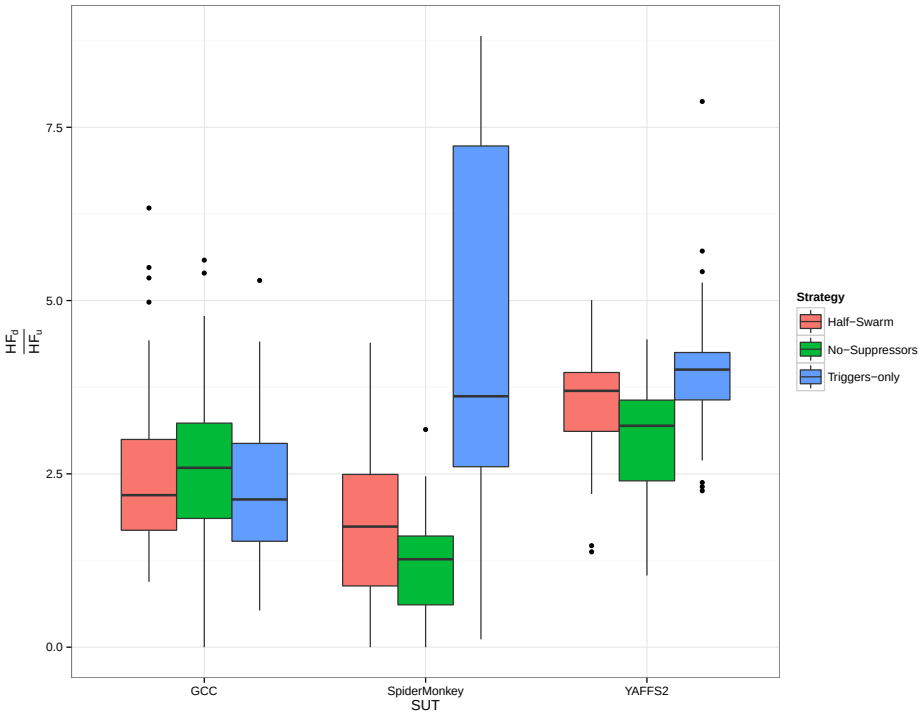

Figure 2.4: Single-target strategies compared.

Observation 2: There is no clear best strategy for single-target testing, though it is clear that adopting Triggers-only may be risky in some settings.

2.6.2 RQ3 and RQ4: Multiple-Target Strategies

For analysis of multi-target directed testing, we use the *average* hitting fraction, i.e. $\overline{HF}$, for comparison of effectiveness of directed testing. Figure 2.5 shows results $\frac{\overline{HF_d}}{\overline{HF_u}}$ with various target set sizes. The most obvious trend is that, while it is effective, the effectiveness of directed testing decreases with an increase in number of targets. For YAFFS, targeted testing always increases the hitting fraction of targets, on average between 1.89x to 2.82x .

In GCC, directed testing improves hitting fraction for 81.2% of all target sets. No-suppressors and Half-swarm strategies improve the hitting fraction of targets in 95.7% of cases. On average, they improve hitting factor between 1.3x to 2.26x.

Directed swarm testing improves the hitting fraction for 73.4% of target sets in SpiderMonkey. The Triggers-only strategy for SpiderMonkey could not generate tests for many targets, due to the complex recursive code generation in `jsfunfuzz` (mentioned earlier) generating test suites for only about 41% of targets. Half-swarm and No-suppressor strategies improve the hitting fraction for 72.2% of their targets, between 1.07x and 1.48x on average, for target set sizes of 10 and 5, respectively.

The result of rank-sum test of effectiveness of multiple-target testing suggests that the Triggers-only strategy does not perform well in generating effective configurations to increase the frequency of coverage for targets[6]. Given the risks seen in single-target testing and the lackluster results here, we believe that Triggers-only may be the least effective strategy, despite its good results for YAFFS2 single-target directed swarm testing. It may be that Triggers-only is simply too extreme: conventional random testing uses all features in every test, and swarm testing can often improve this by reducing the fraction of features by half. Lowering it to the small number of triggers for many targets may simply not match the behavior most random testers are designed to work with, or produce too little complex interaction of software components to provide good testing.

Observation 3: For YAFFS, GCC, and SpiderMonkey, for No-suppressor and Half-swarm strategies, the hitting fraction of at least 95% of target sets increases significantly using directed swarm testing ($p < 0.01$, Wilcoxon rank-sum).

[6]Full statistical test results in tech report.

Figure 2.6 shows the performance of different merge strategies across test subjects, and Figure 2.7 shows how merge strategies affected the number of effective targets (how much merging was possible). Aggressive merging produced consistently very small sets of targets, while Subsumption results are generally closer to Round-robin than to Aggressive. The difference between the Round-robin and Subsumption strategies in hitting fractions was therefore minimal (and in most cases, not statistically significant). The most likely explanation is that when two targets have similar enough triggers and suppressors to merge, for our subjects, testing one target in round robin is likely to "accidentally" target the other target as well. Aggressive merging improved hitting fractions for GCC and SpiderMonkey, but performed poorly for YAFFS2.

2.6.3 RQ5: Actual Fault Detection

In addition to our basic results showing that directed swarm testing can improve the frequency of coverage of targets, we also performed experiments on actual fault detection — the hypothesis that producing more tests hitting a code target will likely find faults involving that code more easily seems obvious, but there could be confounding factors, such as reduction of some other form of test diversity produced by swarm testing. These experiments are based on 7 randomly chosen known (fixed) SpiderMonkey faults. For each of these faults, we targeted the statements in the code commit that introduced the fault. Evaluation was based on comparing 30 minute undirected and directed swarm suites, and counting how many times the fault was detected, on average, over a large (¿ 50) number of trials. For two of the faults, neither directed nor undirected testing ever detected the fault. The commit sizes for the remaining 5 faults were 40, 13, 5, 17, and 15 statements, respectively. Table 2.4 shows detection rates for undirected swarm testing and directed strategies, with the best detection rate for each fault in bold. Triggers-only is omitted from results, due to its difficulties producing valid SpiderMonkey tests, and Aggressive merging did not actually produce any additional merges over those provided by Subsumption. While no single strategy dominated all others, some basic points are clear: first, undirected swarm never had the best detection rate, and had the worst detection rate for 3 of the 5 faults. Second, Round-robin Half-swarm never had the worst detection rate, and had the best detection rate for 2 of the 5 faults, and improved the detection rate compared to undirected swarm testing by an average of 2.56x. Subsumption No-suppressors also always improved on undirected testing. Due to the large number of similar results (most runs did not detect a fault), however, these differences were only statistically significant for Fault #5.

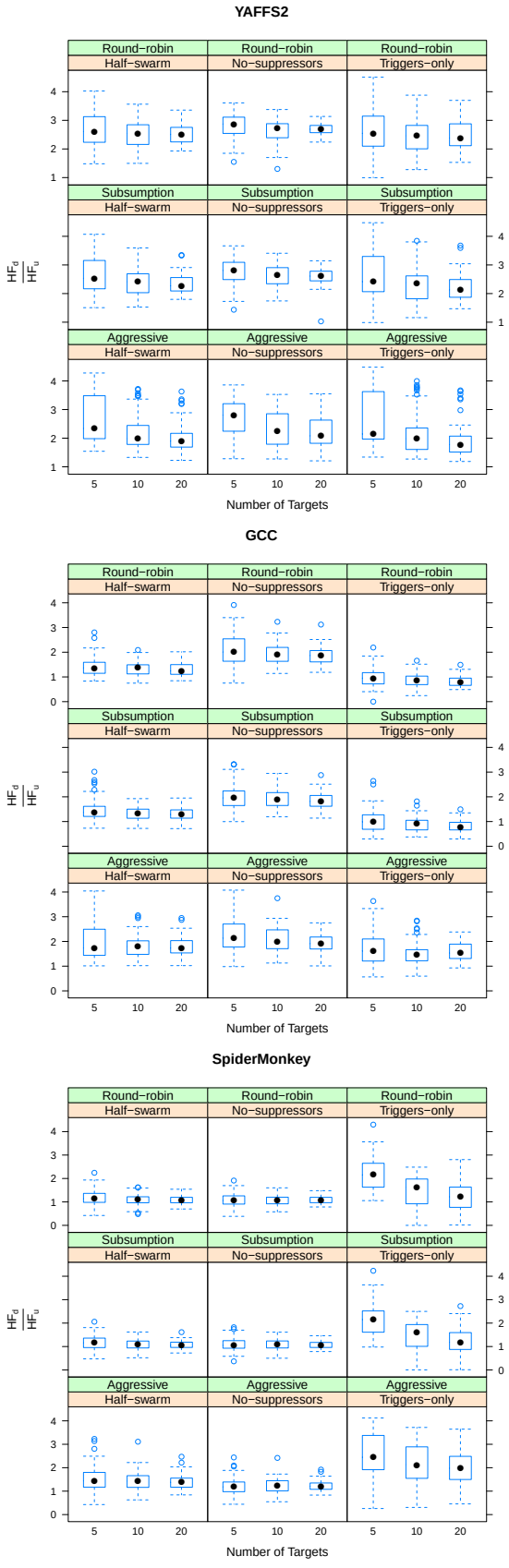

Figure 2.5: Multi-target strategies compared.

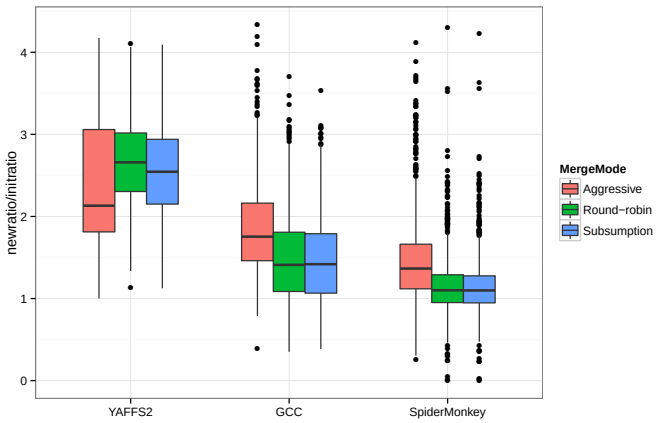

Figure 2.6: Merge strategies over all multi-target strategies.

Table 2.4: Detection rate of actual faults in the test suites generated by each technique in a 30-minute test suite generated by each test strategy.

Test Strategy	Fault #1	Fault #2	Fault #3	Fault #4	Fault #5
Undirected Swarm	13.6	0.07	0.24	0.26	0.07
Round-robin Half-swarm	31.9	0.19	**0.35**	**0.56**	0.29
Round-robin No-suppressors	**34.2**	0.26	0.17	0.46	**0.69**
Subsumption Half-swarm	33.0	0.24	0.12	0.10	0.29
Subsumption No-suppressors	33. 1	**0.31**	0.29	0.31	0.46

Observation 4: Directed swarm testing, for 5 real SpiderMonkey faults, ususally detected real faults much more frequently than undirected testing. Round-robin Half-swarm was arguably the most effective approach.

2.6.4 RQ6: Comparison with Random Testing

We chose to use random testing without swarm configuration as an external evaluation. If directed testing cannot improve the hitting fraction of targets over pure random testing, the applicability of our technique would be questionable. Comparison with other techniques would have little value, as the testing strategies in most search-based and symbolic testing methods we are aware of essentially aim to cover each code target once, not to maximize frequency of coverage, unless the target coincidentally is hit when covering other targets. Frequency of coverage

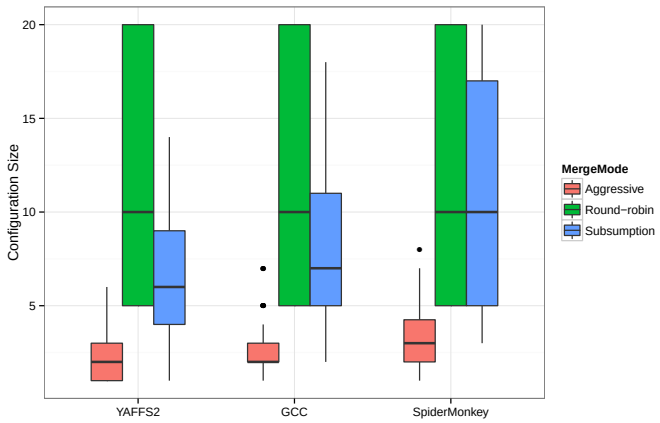

Figure 2.7: Number of targets after merging, by merge strategy.

is therefore a largely meaningless metric for these methods, while it is often used by engineers evaluating random testers (if a random tester hits a code target very infrequently it can be seen as a problem with the tester) [30]. We compared the average hitting fraction of targets in single-targeted experiments over all strategies for each SUT (HF_d), with the average hitting fraction in test suites generated by traditional random testing (HF_r) under the same time budget as in the single-target directed swarm experiments, with the same number of trials as in the earlier single-target experiments. Figure 2.8 illustrates the results. We used a paired t-test between HF_d and HF_r. Table 2.5 summarizes the results, showing average hitting fractions, confidence intervals on the effect size, and p-values.

Table 2.5: The result of t-test comparing the hitting fraction of targets in directed random testing and random testing without swarm.

SUT	HF_r	HF_d	confidence-interval	p-value
YAFFS2	0.671	0.819	(0.126,0.170)	0.0000
GCC	0.342	0.425	(0.053,0.113)	0.0000
SpiderMonkey	0.198	0.276	(0.059,0.097)	0.0000

Observation 5: Directed swarm testing significantly ($p < 0.01$) increases hitting fractions over pure random testing.

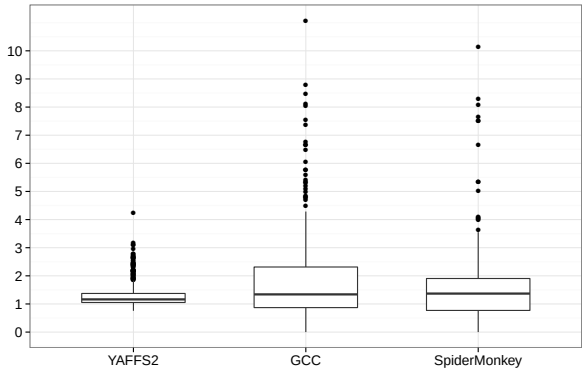

Figure 2.8: $\frac{HF_d}{HF_r}$, random vs. directed.

2.7 Threats to Validity

Threats Due to Sampling Bias: Our results are based on results from only three large open source software programs. While we believe that these programs are well tested examples of real-world programs, there is a possibility that they are not representative. All our subjects are "systems" software in C, for example. Generalizing to other languages and types of code may be unwarranted.

Limited External Evaluation: We used pure random testing as our external evaluation. We are aware that there are other techniques that aim to cover particular code targets, most notably search-based techniques and symbolic execution. However, to our knowledge, all of these tools aim to produce a single test covering each target, not a set of highly diverse tests that frequently cover the target(s); even tools aiming to test code patches (e.g. KATCH [43]) aim to hit targets only once.

2.8 Discussion

While our results generally support the effectiveness of directed swarm testing, it is surprising how difficult it is to identify a single best strategy for directed swarm testing. Triggers-only is likely ineffective, but choosing between Half-Swarm, No-Suppressors, Round-Robin, Subsumption, and Aggressive strategies is not simple. In part we attribute this to the underlying complexity of what is happening in (directed) swarm testing: each configuration defines a (usu-

ally effectively infinite) set of tests. This is, of course, the point of random testing, that an unbounded number of diverse tests can be generated, using all available testing budget.

Swarm testing improves random testing in many cases, in the long run, by increasing the diversity of generated tests. This diversity can come with a price, however: for a fixed testing budget, because swarm testing improves diversity, the hitting fraction for many individual targets will be lower than for pure random testing (when swarm testing increases overall coverage, this is almost required — hitting more targets means hitting each target less often [31]). In fact, we noticed that comparing hitting fractions for undirected swarm testing and pure random testing, we often saw better hitting fractions for pure random testing, despite the fact that fault detection and overall coverage tend to show swarm testing performing much better for reasonably-long test runs [31]. Configuration strategy not only determines individual test behavior, but determines how quickly coverage saturates due to (lack of) diversity of tests created. Swarm testing produces very diverse tests; random testing without swarm configuration produces much less diverse tests. Our directed swarm testing strategies introduce a large number of choices in between these extremes, with a given focus [26]. Our experiments show that a variety of configuration methods can improve hitting fractions, but understanding how to best choose a strategy for, e.g., short vs. long budget directed testing is an open question we would like to address. However, the primary aim of directed swarm testing will typically be to detect faults quickly. One reasonable approach is to extend the diversity-centric ideas of swarm testing to strategy selection, and run in parallel directed tests for a change set using all of the viable strategies (e.g., all but Triggers-only).

2.9 Related Work

The most closely related work is our previous work introducing swarm testing [31] and the notions of triggers and suppressors [25]. We expand on that work by using the concepts introduced to enable a practical way to generate focused random tests.

There are several approaches for generating a test case that covers a chosen source code target once. Of these, search-based testing [34, 44] and (dynamic) symbolic execution [22, 59] are the most notable ones. Symbolic execution [40] formulates an execution path in the program as a constraint formula problem and generates inputs that satisfy the path conditions and hence cover the target. Dynamic symbolic execution improves the scalability of pure symbolic execution by using information from concrete executions to replace over-complex constraints, simplifying problems of handling, e.g. system calls and pointers [22]. Search-based testing reduces the

problem of covering a particular entity in the program to a search problem and uses techniques such as genetic algorithms and hill climbing, to solve that problem [34, 44].

There are many previous efforts to improve random testing. Randoop [46] generates tests for object-oriented programs by calling random APIs, but uses feedback to guide test sequence creation. Nighthawk [6] uses genetic algorithms on top of a random tester to modify the configuration of the random tester to optimize it for a given goal (i.e., fitness function). Adaptive random testing [11, 12] aims to improve random testing by using a distance measure to select more uniformly distributed tests, though its effectiveness in practice has been criticized [8]. ABP-based testing uses reinforcement learning to guide test generation [29].

To our knowledge, none of these approaches are applicable to the problem we address. First, we believe that our approach is the only attempt to produce a large set of diverse tests (due to random variation, in our case, but any type of diversity would be useful) that cover certain code targets with high frequency. While symbolic execution and search-based testing may be helpful for producing tests targeting a given element in source code, they simply attempt to hit the target, not produce many tests hitting the target in various ways. Moreover, these approaches are not always easy to apply to complex SUTs (such as a production quality compiler that takes as input full programs in a complex language), and symbolic execution in particular is often far less efficient than random testing [64]. Symbolic execution unfortunately simply fails to scale to very large systems with complex input, in many cases, or requires seed tests. The approach proposed in this paper is often trivial to apply to existing random test generators for complex software systems and, like pure random testing, has extremely low overhead (collecting coverage information on some random test runs is the only real cost, and this is only paid during data collection, not during new testing runs). While other methods are suitable for generating a *single test* targeting specific code (and this is their common usage), the high cost of each test generated by many methods might make them unsuitable for our purposes of high frequency of coverage in diverse tests, even if some variation were proposed allowing the generation of multiple tests for a target.

2.10 Conclusions and Future Work

In this paper we demonstrate that using collected statistics on code coverage and swarm testing, it is possible to produce focused random tests — truly random tests that nonetheless target specific source code. While results for the various strategies for directed swarm testing vary, in general

the method is able to increase the frequency with which tests cover targeted code by a factor often more than 2x, and sometimes up to 8 or 9x. This approach is readily applicable to existing, industrial-strength random testing tools for critical systems software, and therefore out-of-the-box scalable to applications such as testing production compilers and file systems.

In conjunction with existing regression suites and other methods, we hope that applying some element of "regression testing" (targeting code changes) to highly diverse and cost-effective random testing can make it easier to find faults in changed or otherwise suspicious parts of complex systems. For example, if static analysis indicates that a source code line may have a bug, but the analysis technique is subject to false positives, it may be useful to subject such lines to further scrutiny with targeted tests. If mutation testing reveals that many mutants of certain code lines survive an existing test suite or a large number of random tests, directed swarm testing can be used to produce random tests that have more chance of killing these mutants, for inclusion in regression tests. Targeting source code that is very infrequently covered during extensive random testing, but covered enough to provide a basis for statistical estimation of triggers and suppressors may lead to covering code that the seldom-covered code dominates in the CFG, improving the overall effectiveness of large-scale random testing. Targeting faults, rather than source code lines, can help improve suites for fault localization, by producing more failing tests to analyze. We believe there may be further practical applications of the combination of test suite statistics and variation in test case configurations. The changes in effectiveness of directed swarm testing, depending on the strategy chosen for balancing focus and diversity also show the difficulty of understanding complex testing systems.

2.11 Acknowledgements

The authors thank Scott Davies for discovering the equivalence to optimal tuple matching, and John Regehr, Darko Marinov, Milos Gligoric, Josie Holmes, and Mihai Cobodan for useful discussions. A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786.

Evaluating Non-adequate Test-Case Reduction

Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce.

In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).
ACM, New York, NY
pp 16-26.

Chapter 3: Evaluating Non-adequate Test-Case Reduction

3.1 abstract

Given two test cases, one larger and one smaller, the smaller test case is preferred for many purposes. A smaller test case usually runs faster, is easier to understand, and is more convenient for debugging. However, smaller test cases also tend to cover less code and detect fewer faults than larger test cases. Whereas traditional research focused on reducing *test suites* while preserving code coverage, recent work has introduced the idea of reducing individual *test cases*, rather than test suites, while still preserving code coverage. Other recent work has proposed non-adequately reducing test suites by not even preserving all the code coverage. This paper empirically evaluates a new combination of these two ideas, *non-adequate reduction of test cases*, which allows for a wide range of trade-offs between test case size and fault detection.

Our study introduces and evaluates $C\%$*-coverage* reduction (where a test case is reduced to retain at least $C\%$ of its original coverage) and N*-mutant* reduction (where a test case is reduced to kill at least N of the mutants it originally killed). We evaluate the reduction trade-offs with varying values of $C\%$ and N for four real-world C projects: Mozilla's `SpiderMonkey` JavaScript engine, the `YAFFS2` flash file system, `Grep`, and `Gzip`. The results show that it is possible to greatly reduce the size of many test cases while still preserving much of their fault-detection capability.

3.2 Introduction

Smaller test cases are, in many ways, preferable to larger test cases. For example, smaller test cases tend to run faster, which can improve the efficiency of running test suites [24], i.e., sets of of individual test cases. Smaller, simpler test cases are also easier to understand and enable more effective debugging. This was the initial motivation for delta-debugging [63]—a technique for reducing the size of failing test cases. Because of the advantages of smaller test cases, random test generation is often combined with delta-debugging, making research on effective reduction techniques itself an important topic [14,30,42,49]. Test suites with small test cases (that focus on

separate functional properties) also make it possible for test-case selection [19] and prioritization to operate more effectively than when applied to test suites mostly consisting of large, complex test cases.

While smaller test cases have advantages, it is also true that smaller test cases, all else being equal, detect fewer faults than larger test cases [5]. The trade-off between size and effectiveness for individual test *cases* is similar to the trade-off between smaller and larger test *suites*. Researchers have extensively studied test-*suite* reduction [33, 35, 51, 54, 61], which removes entire test cases from test suites.

The problem of *test-suite reduction* is to reduce a given test suite while preserving most of its fault-detection capability. Various techniques have been proposed, and many are summarized in a survey by Yoo and Harman [61]. Test-suite reduction trades off reduced fault-detection capability (most often measured by the number of killed mutants) for reduced test-suite size (typically measured by the number of test cases). Traditional techniques *completely* preserve some property of a test suite, e.g., its code coverage, while removing test cases that are redundant and do not contribute to that property. Recently, we evaluated non-adequate test-suite reduction [54] that only *partially* preserves the property of interest, e.g., preserves 90% of code coverage.

The problem of *test-case reduction* is to reduce an individual test case while preserving most of its fault-detection capability. Reducing a test case essentially requires "slicing and dicing" the atomic parts that make the test case. For example, if a test case is a unit test composed of a sequence of function calls, reduction usually involves removing function calls. If a test case is defined by an input file, reduction can involve removing characters from the file. Note that measuring the *size* of a test case is inherently project-specific, depending entirely on the semantics of test cases, whereas the size of test suites can be defined in a project-agnostic way as the number of test cases in the test suite (though not perfectly correlated with the time to execute the tests). While test-suite reduction has been studied in depth at least since 1993 [35], test-case reduction research is much more recent.

Zeller and Hildebrandt proposed delta-debugging [63], the best known test-case reduction technique, usually applied to reduce a failing test case to a minimal test case that still fails. Recently, we proposed *cause reduction* [24, 28] as a generalization of delta-debugging, and used it to reduce a (passing or failing) test case while preserving its original coverage. Cause reduction *completely* preserves coverage: the reduced test case has to cover all the code elements that the original test case covered (and can potentially cover even more). We call such reductions *adequate* because they preserve 100% of some property. Note that "adequate" in our context

refers to the relationship between the reduced and original test cases, although the original test case itself may provide far from adequate code coverage.

The utility of non-adequate reduction for test suites [54] naturally suggests that non-adequate reduction may be useful for test cases as well. Non-adequate reduction, either for test suites or test cases, greatly enlarges the number of points to explore in trading off size and fault-detection capability. For test cases, requiring adequacy limits how much size can be reduced (some test cases cannot be reduced substantially without sacrificing at least some coverage or killed mutants) and increases the time required to reduce test cases (because searching for an adequate reduction is often harder than finding a "good enough" reduction).

Combining the recent ideas of non-adequate reduction for test suites [54] and adequate reduction for test cases [24,28], this paper empirically evaluates a new combination: *non-adequate reduction for test cases*. To the best of our knowledge, ours is the first such evaluation.

Specifically, we evaluate $C\%$*-coverage* reduction (where a test case is reduced to retain at least $C\%$ of its original coverage) and N*-mutant* reduction (where a test case is reduced to kill at least a given set of N mutants it originally killed). Both reductions reduce a larger test case to a smaller test case while only *partially* preserving some property. Hence, we call these reductions "non-adequate" because they do not necessarily preserve completely either the code coverage or all mutants killed. However, the reduced test case *could*, in theory, cover code elements or kill mutants that the original test case does not, even if the reduced test case does not cover all code elements or kill all mutants that the original test case did; in fact, the reduced test case can even cover *more* code or kill *more* mutants.

Non-adequate test-case reduction further generalizes previously proposed test-case reductions. By parameterizing the level to which the reduced test case needs to preserve a property, we allow more freedom to explore trade-offs between size reductions and preservation of fault-detection capability [54, 62]. For example, cause reduction [24, 28] becomes just a special case of our $C\%$-coverage with $C = 100$. Preserving to kill only one mutant that encodes some fault (N-mutant with $N = 1$) can mimic delta-debugging. At the other extreme, setting N to equal the total number of all mutants originally killed results in a very strict test-case reduction that preserves *all* mutants killed; however, such reduction may be prohibitively expensive to perform (and would likely provide very little reduction unless test cases have excessive redundancy), so our evaluation concerns only small values for N.

We evaluate non-adequate test-case reduction on four real-world C projects: Mozilla's `SpiderMonkey` JavaScript engine, the `YAFFS2` flash file system, `Grep`, and `Gzip`. We used manual `Grep` test

cases, and automatically generated test cases for the other projects. We evaluate $C\%$-coverage for various levels of $C\%$, from 70% to 100%. We evaluate N-mutant with (1) randomly selected mutants for various values of N from 1 to 32, and (2) mutants that are hard to kill based on the minimal mutant set [3]. We measure size reduction, code coverage, and mutants killed, with the latter two[1] used as proxies for fault-detection capability. Our results show that in many cases, non-adequate test-case reduction can substantially reduce the size of the given test cases while still preserving considerable fault-detection capability. Perhaps most interestingly, when performing $C\%$-coverage reduction, the largest gain in size reduction for all cases comes when $C\%$ changes from 100% to 95%; the gain is typically twice as large as for any other $C\%$ change. This gain does *not* result in a similarly large loss in mutation detection. In brief, simply giving up on perfection enables a larger reduction in size than the associated reduction in effectiveness. Additionally, preserving even a small number N of mutants killed usually indirectly preserves a large fraction of all other mutants killed, often more than 70%.

This paper makes the following contributions:

- **Novel test-case reduction approach:** We define two types of *non-adequate test-case reduction*: $C\%$-coverage and N-mutant reduction.
- **Evaluation of reduction trade-offs:** Using four real-world C projects, we evaluate the relationship between the size reductions obtained with varying parameters for these reductions, and the code coverage and killed mutants for reduced test cases relative to the original, unreduced test cases.

3.3 Non-adequate Test Reduction

We next describe our test-case reductions in more detail. We use t_o to denote the original test case and t_r to denote the reduced test case. We use t to denote an arbitrary test case, $Cov(t)$ to denote the set of statements[2] covered by t, $Mut(t)$ to denote the set of mutants killed by t, $|S|$ to denote the cardinality of the set S, and $Size(t)$ to denote the size of t. Measuring the size of a test case is specific to the project or the format of test cases; Section 3.5.1.1 precisely defines size for the projects used in our evaluation. Conceptually, we define size as the number

[1] Although they are not ideal proxies, code coverage is often used by developers to evaluate quality of test cases and both are commonly used to evaluate test cases in research.

[2] While we present and evaluate $C\%$-coverage only for statement coverage, it can generalize, e.g., to branch coverage.

of atomic *parts* that a test case has. The parameterized nature of parts is taken from the original delta-debugging work [63]. In some projects, parts are function calls; in other projects, they are lines or characters in a file/string; and in rare cases, they may be much more complex, e.g., defined by a grammar. For example, reduction of test cases that are computer programs (e.g., an input to a compiler) [49] often relies on a semantically involved notion of part.

The high-level goal of test-case reduction is to produce a reduced test case t_r with size smaller than the size of t_o, i.e., $Size(t_r) < Size(t_o)$ (and ideally $Size(t_r) \ll Size(t_o)$), such that t_r still retains (either completely or partially) some desirable property of t_o. That is, for some notion of "quality", t_r has similar quality to t_o. t_o itself may have good or bad quality, but t_r should not have much worse quality than t_o. While in principle the reduction process can stop at various steps (and in the limit, even the original test case can be considered a reduced version of itself), we are interested in so called "1-minimal" test cases [63] where no single part of t_r can be removed without losing some desired property.

3.3.1 Reduction Algorithm

The test-case reduction algorithm we use is derived from the original delta-debugging [63] algorithm, and we modify it to support non-adequate test-case reduction. Delta-debugging takes as input a *failing* test case and reduces it by removing parts that are not relevant for the failure. A generalized algorithm for cause reduction [24, 28] extends delta-debugging to reduce a test case with respect to *any* property, not just failure, that can be detected when running the test case. The most direct application of cause reduction is to *completely* preserve code coverage.

At a high level, the delta-debugging algorithm (described in detail by Zeller and Hildebrandt [63] and extended in the work on cause reduction [24, 28]) iteratively splits a test case into multiple candidate test cases. At each of these steps, the algorithm checks if any candidate satisfies the desired property (which, in traditional delta-debugging, is whether the test case fails). If there is a satisfactory candidate, it becomes the new base test case to be reduced further in the future steps. If no candidate is satisfactory, the granularity for splitting is increased, until the algorithm determines that the test case is 1-minimal: removing any single part produces a test case that does not satisfy the property. In this paper, we further generalize delta-debugging and cause reduction by allowing the candidate test case to only *partially preserve* some property.

3.3.2 $C\%$-Coverage Reduction

We relax the requirement from cause reduction [24,28]—that the reduced test case t_r preserve all code coverage obtained by the original test case t_o—with the requirement that t_r preserve at least $C\%$ of coverage obtained by t_o. Reducing large test cases to preserve all (statement) coverage can be prohibitively expensive. For example, we previously reported that cause reduction of a single test case for the GCC compiler could take days [24,28]. Moreover, preserving 100% of the coverage *may not be necessary*, because a test case that preserves less may still have acceptable quality. Hence, we propose $C\%$-coverage reduction:

Definition 1 *$C\%$-coverage test-case reduction produces a reduced test case t_r that covers at least $C\%$ of the statements covered by the original test case t_o:*

$$\frac{|Cov(t_r) \cap Cov(t_o)|}{|Cov(t_o)|} \geq C\%$$

Note that the percentage is determined by the coverage of the *original* test case and *not* by coverage over *all* statements in the code under test. The property is not $\frac{|Cov(t_r)|}{|Cov(t_o)|} \geq C\%$, because t_r could then end up covering statements unrelated to those covered by t_o. Coverage-based cause reduction [24, 28] can be (re)defined as $C\%$-coverage with $C = 100$: $|Cov(t_r) \cap Cov(t_o)|/|Cov(t_o)| = 100\%$, or equivalently $Cov(t_r) \supseteq Cov(t_o)$. $C\%$-coverage does not impose any requirements over statements *not* covered by the original test case: the reduced test case may or may not cover those statements. Also, $C\%$-coverage does not (directly) require any relationship between $|Cov(t_r)|$ and $|Cov(t_o)|$, so it can even happen that $|Cov(t_r)| > |Cov(t_o)|$ if t_r covers some statements that t_o does not cover.

3.3.3 N-Mutant Reduction

We define N-mutant reduction in a similar fashion, but with three important differences: (1) N-mutant uses killed mutants instead of covered statements; (2) N-mutant preserves the ability of a test case to kill an absolute number N of mutants rather than a relative ratio of mutants; and (3) N-mutant considers the *same* set of selected mutants for all steps of the reduction algorithm:

Definition 2 *N-mutant test-case reduction produces a reduced test case t_r that kills a specific set of N mutants selected from the set of $Mut(t_o)$, where typically $N \ll |Mut(t_o)|$.*

The difference (3) from $C\%$-coverage is largely motivated by the cost of determining the complete set of mutants killed for every candidate test case at each step of the reduction algorithm. We did initially experiment with allowing the set of mutants to change, while requiring only that the number of mutants be preserved through reduction steps be at least N. However, by allowing the algorithm to only preserve at least any N mutants, it can be necessary to run a large number of mutants at each step of the reduction algorithm (until at least N mutants are killed or *all* mutants are run and N are not killed). As a result, the time to perform non-adequate test-case reduction was often prohibitively long. Again, we only require the selected N mutants to be a subset of $Mut(t_o)$. Mutants other than those in the selected set may or may not be killed by t_r.

3.4 Metrics

We describe three metrics for evaluating the effectiveness of test-case reduction: Size Reduction Rate (SRR), Coverage Preservation Rate (CPR), and Mutant(-killing) Preservation Rate (MPR). We define all metrics such that higher values are better and values are normalized to the range 0%–100%.

3.4.1 Size Reduction Rate (SRR)

The goal of test-case reduction is to reduce the size of a test case. As such, it is important to measure how much smaller the reduced test case is compared to the original test case. Recall that $Size(t)$ denotes the size of a test case t, i.e., the number of the atomic parts that the test case has.

Definition 3 *For an original test case t_o and its reduced test case t_r, Size Reduction Rate (SRR) is:*

$$SRR(t_o, t_r) = \frac{Size(t_o) - Size(t_r)}{Size(t_o)}$$

A higher SRR is desirable as it indicates that more parts have been removed from the test case, resulting in a smaller reduced test case.

3.4.2 Coverage Preservation Rate (CPR)

Our reduction is non-adequate test-case reduction, so we need some metrics to measure how much fault-detection capability the reduced test case loses compared to the original test case. Structural code coverage, although not an ideal proxy for fault-detection capability [20, 23, 37], is commonly used to evaluate the quality of test cases: the more code a test case covers, the higher the chance it can detect a fault. We therefore use statement coverage as one way to evaluate quality. Recall that $Cov(t)$ denotes the set of statements covered by a test case t.

Definition 4 *For an original test case t_o and its reduced test case t_r, Coverage Preservation Rate (CPR) measures the ratio between the number of statements covered by both t_r and t_o and the number of statements covered by t_o:*

$$CPR(t_o, t_r) = \frac{|Cov(t_r) \cap Cov(t_o)|}{|Cov(t_o)|}$$

A higher CPR is desirable as it indicates the reduced test case covers a larger subset of statements covered by the original test case. Note that while a reduced test case can potentially cover more statements than the original test case, CPR is limited to 100% as it considers only the statements covered by t_o.

3.4.3 Mutant Preservation Rate (MPR)

MPR is essentially the same as CPR, except measured with respect to mutants killed, not statements covered:

Definition 5 *For an original test case t_o and its reduced test case t_r, Mutant Preservation Rate (MPR) measures the preservation of mutants killed by t_r relative to the mutants killed by t_o:*

$$MPR(t_o, t_r) = \frac{|Mut(t_r) \cap Mut(t_o)|}{|Mut(t_o)|}$$

A higher MPR is desirable as it indicates the reduced test case is better at killing mutants among those that the original test case kills. Like CPR, MPR is relative to the original test and cannot exceed 100%.

Table 3.1: Four projects used in our evaluation and some statistics of their test cases and mutants

project	NCLOC	# test cases	definition of an atomic part	# mutants	min killed	max killed	test pool	# minimal mutants
`SpiderMonkey`	81,920	99	A statement of JavaScript program	69,067	8101	12825	850	256
`YAFFS2`	10,356	99	One API call	15,046	2071	3439	1000	57
`Grep`	8,433	112	A character in command-line arguments	7,591	19	993	840	99
`Gzip`	5,129	73	A byte in the input file	7,175	1813	2046	1000	32

3.4.4 Reduction Requirements vs. Metrics

Although both of the reduction algorithms and the metrics are based on coverage and mutants, note that the requirements for reduction are *not* the same as the metrics used to evaluate the reduced test cases. Therefore, we cannot *a priori* tell how high or low the metrics will be for all reductions. For $C\%$-coverage reduction, we know that CPR will be at least $C\%$, but it could be much higher (up to 100%), and MPR could in theory range from (literally) 0 to 100%. For N-mutant reduction, we know that MPR will be at least $N/|Mut(t_o)|$, but it could be much higher (in our experiments, even when $N/|Mut(t_o)| < 1\%$, MPR can be quite high), and CPR could range from almost 0 to 100%.

3.5 Evaluation Methodology

We describe the projects, test cases, and mutants used in our evaluaton (Section 3.5.1) and the experimental setup (Section 3.5.2). Our experiments ran on a high-performance cluster of commodity computing nodes; each node had 6–12 2.6Ghz Intel Xeon cores.

3.5.1 Projects

Table 3.1 lists the projects used in our evaluation. We tabulate the project name, the number of non-comment lines of code, the number of test cases used in our evaluation, what an atomic part is, the total number of mutants used, and the minimum and maximum number of mutants killed by each test case. The last two columns are the number of tests in a randomly generated test pool for each project and the number of minimal mutants determined for each project; these last two columns are metrics relevant for our analysis involving minimal mutants (Section 3.5.1.3). We use four small- to medium-size C projects: `SpiderMonkey` is Mozilla's JavaScript engine, `YAFFS2` is a popular flash file system used in the early Android versions, `Grep` is the standard Unix utility for searching files, and `Gzip` is the standard Unix utility for

compressing/decompressing files.

3.5.1.1 Test Cases

We use automatically generated test cases for `SpiderMonkey`, `YAFFS2`, and `Gzip`, and we use manually written test cases for `Grep`. The `SpiderMonkey` test cases are JavaScript programs randomly generated using the highly successful `jsfunfuzz` [52] fuzzer. The `YAFFS2` test cases are sequences of API calls to the file system, randomly generated using a publicly available test generator for `YAFFS2` that has been used by several research projects on test generation [13, 24, 28]. The `Gzip` test cases are files that have 500 to 3,500 random bytes. For `Grep`, we use the manually written test cases obtained from `SIR` [18]; each test case consists of command-line arguments to `Grep`.

How best to measure the *size* of a test case is an open question in software testing research. Researchers use a variety of metrics, such as the number of API calls, execution time, or number of assertions. As in our previous work [24, 28], we define size as the number of atomic *parts* of atest case. The concrete part differs from one project to another, as summarized in Table 3.1. A part is a JavaScript code fragment in the generated program for `SpiderMonkey`, one API call in the generated sequence of API calls for `YAFFS2`, a character in the command-line arguments for `Grep`, and simply one byte in the input file for `Gzip`.

We limit the size of generated test cases to enable experiments to finish in a reasonable amount of time. Time complexity of the basic delta-debugging algorithm is quadratic [63] in the number of parts in a test case. For `SpiderMonkey`, `YAFFS2`, and `Gzip`, we control the number of parts based on the specific limits from our initial experiments, trying to finish most test-case reductions within 30 minutes. In particular, we limit each `SpiderMonkey` test case to be exactly 200 lines of JavaScript code, each `YAFFS2` test case to be a sequence consisting of exactly 200 API calls, and each `Gzip` test case to be a file consisting of at most 3,500 bytes. For `Grep`, we use *all* the test cases manually written by others [18], and we do not limit their sizes; the largest test case for `Grep` has 146 characters in the command-line arguments.

3.5.1.2 Mutants

We use a mutation-testing tool for C code developed by Andrews *et al.* [7] and used in many previous studies. Quoting [7], the tool provides the following four classes of mutation operators:

"(1) Replace an integer constant I by 0, 1, -1, $((I)+1)$, or $((I)-1)$; (2) Replace an arithmetic, relational, logical, bit-wise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class; (3) Negate the decision in an if or while statement; and (4) Delete a statement."

Each mutant was compiled with *GCC* using the highest optimization `-O3` and compared with other binaries to avoid trivially equivalent mutants [48]. About 15% of the generated mutants were found to be trivially equivalent. Table 3.1 shows the number of mutants for each project and the minimum and maximum number of mutants killed by each test case. A mutant is considered killed if its output (including `stdout`, `stderr`, and produced files) differs from the output of the original code.

3.5.1.3 Minimal Mutants

To evaluate N-mutant reduction, we use two methods to select mutants. The first method is simple random sampling: we select N mutants from the set of mutants killed by the test case. In the second method, we wanted to alleviate the impact of redundant and trivial mutants on the results. Redundant mutants are those mutants that are semantically equivalent to one another, albeit syntactically different. Trivial mutants are those mutants that are killed by a majority of test cases. The impact of these two kinds of mutants can be alleviated by using *minimal mutant sets* introduced by Ammann *et al.* [3].

A minimal mutant set is computed based on an original set of killed mutants and a test suite, The first step is to construct a minimal test suite from the original test suite, i.e., a subset of the original test suite that kills all the mutants killed by the original test suite. Removing any test case from the minimal test suite means failing to kill some mutant. Given a minimal test suite, a minimal mutant set is the smallest subset of mutants from the original mutant set such that killing all the mutants from the minimal mutant set (using the minimal test suite) also kills all the mutants from the original set of killed mutants.

We generated minimal mutant sets for projects as follows: first, we generated a large test pool of random test cases for each project. We used these larger pools because minimal mutants require (almost) adequate test suites to ensure that useful mutants are not removed. Then, we obtained the complete set of mutants killed by each test pool. We minimized each test pool with respect to its corresponding project's set of mutants to obtain the minimal set of test cases from the test pool that kill all those mutants, using a greedy test-suite reduction algorithm [61].

Using this minimal set of tests, we minimized the mutant set to obtain the minimal mutant set. Table 3.1 shows the number of test cases in these pools and the sizes of the minimal mutant sets. We later compare randomly selected mutants with minimal mutants by reducing the test cases taken from the large test pool for each project.

3.5.2 Experimental Setup

For $C\%$-coverage, we perform experiments with the non-adequacy value C chosen from the set $\{70, 80, 90, 95, 100\}$. For each original test case, we create a reduced test case that preserves at least $C\%$ of the statements covered by the original test case. We use GCov to obtain the set of statements covered by each test case.

For N-mutant, we perform experiments with the non-adequacy value N chosen from the set $\{1, 2, 4, 8, 16, 32\}$. For each original test case, we first determine what mutants the test case kills and then randomly select N of those mutants (for a small number of test cases that kill fewer than N mutants, we use all mutants) to create a reduced test case that preserves these N selected mutants. To compare randomly sampled mutants with the harder to kill minimal mutants, we take each test case from the minimal test suite (constructed from the large test pool is described in Section 3.5.1.3) and reduce the test case while preserving one randomly selected mutant and then reduce it to perserve the one mutant ($N = 1$) from the minimal mutant set that the test case uniquely contributes to the minimal mutant set. If a test case kills no mutants in the minimal mutant set, we do not reduce the test case at all.

Performing test-case reduction can take a long time for some test cases. We limit reduction to 30 minutes per test case. We observed that N-mutant test-case reduction starts having many timeouts when N gets greater than about 40, so we restrict our choices of N to values less than 40. The experiments ignore test cases whose reduction times out.

For each reduced test case, we further generate three randomly reduced test cases that have *exactly the same size* as the reduced test case. We create such a randomly reduced test case by starting from the original test case and iteratively choosing to remove (uniformly randomly selected) one part at a time until the resulting test case has the same number of parts as the reduced test case. We perform random test-case reduction merely as some kind of baseline to show the benefits of preserving benefits of a test case; we do not actually recommend actually using random test-case reduction in practice.

3.6 Research Questions

Our evaluation addresses the following questions about the effects of non-adequate test-case reduction:

- RQ1: How much are test cases reduced (SRR)?
- RQ2: How much are code coverage and mutants killed preserved (CPR and MPR)?
- RQ3: How do SRR, CPR, and MPR trade off?
- RQ4: How do CPR and MPR for our approaches compare to CPR and MPR for *random* test-case reduction?

3.6.1 RQ1: SRR

Figures 3.1 and 3.2 summarize the results for SRR on the test cases reduced using $C\%$-coverage and N-mutant, respectively. For each project and level of C and N, the boxplots show the distribution of SRR. From the figures, we see that both approaches can greatly reduce the size of test cases. In most configurations, the median SRR for all test cases reduced using either $C\%$-coverage or N-mutant is greater than 50%: the size of a reduced test case is usually less than half the size of the original test case. For both reductions, `Grep` behaves somewhat differently, with median SRR. The likely cause is the small size of test cases in `Grep`: most have < 100 characters.

SRR decreases when C or N increases, as expected. We emphasize that SRR for $C = 100$ is particularly low compared with SRR for other values; allowing coverage to miss even a small set of statements increases SRR substantially. For example, for `Grep`, the median SRR for $C = 100$ and $C = 95$ differ by over 30pp[3].

3.6.2 RQ2: CPR and MPR

Figures 3.3 and 3.4 summarize the results for CPR on the test cases reduced using $C\%$-coverage and N-mutant, respectively. CPR is, of course, always at least as high as the $C\%$ value given to the reduction. From Figure 3.3, for `SpiderMonkey` and `YAFFS2`, CPR is almost exactly the

[3]The "pp" metric (from "percentage points") represents differences in values that are already expressed as percentages.

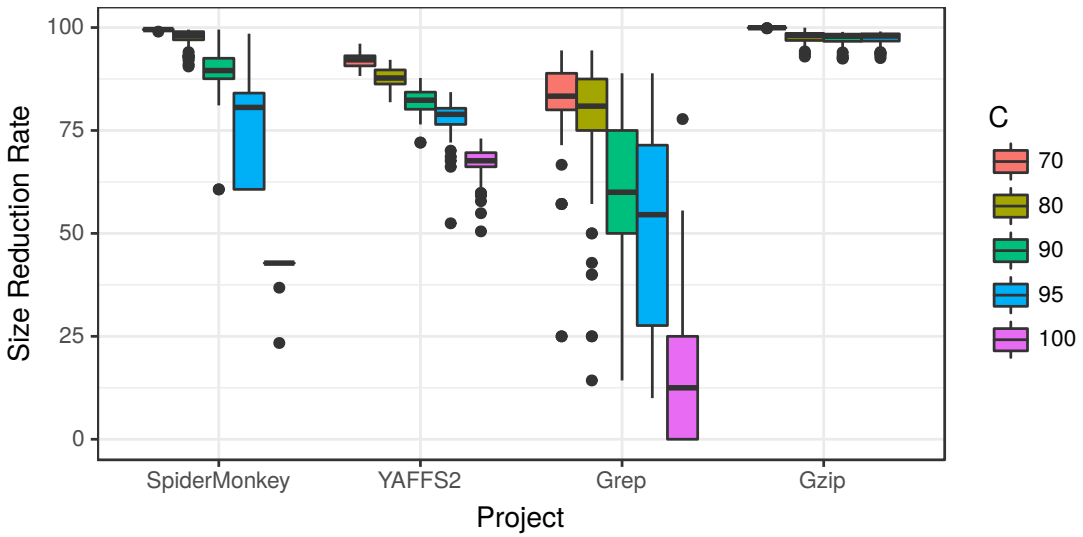

Figure 3.1: SRR for $C\%$-coverage

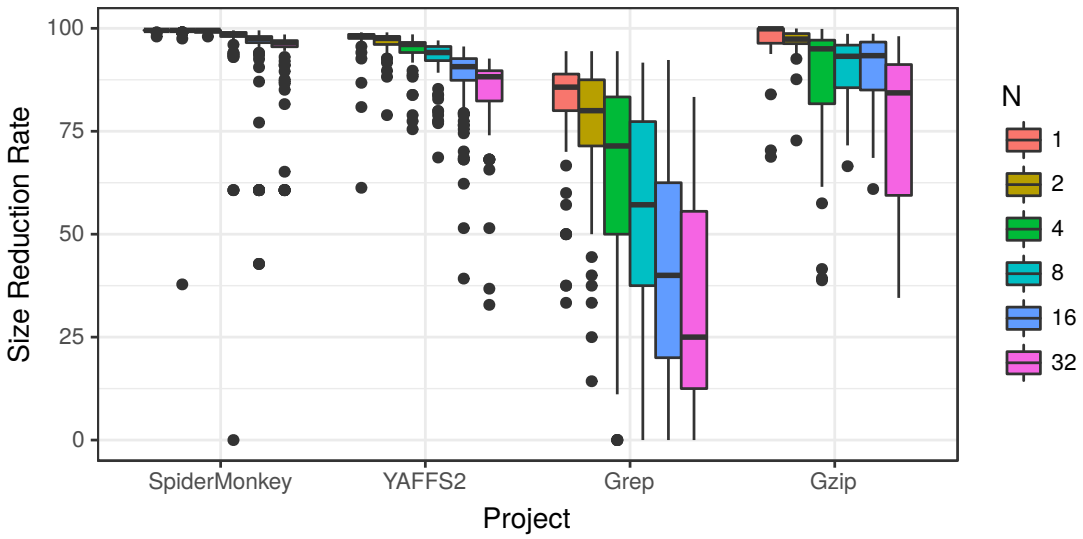

Figure 3.2: SRR for N-mutant

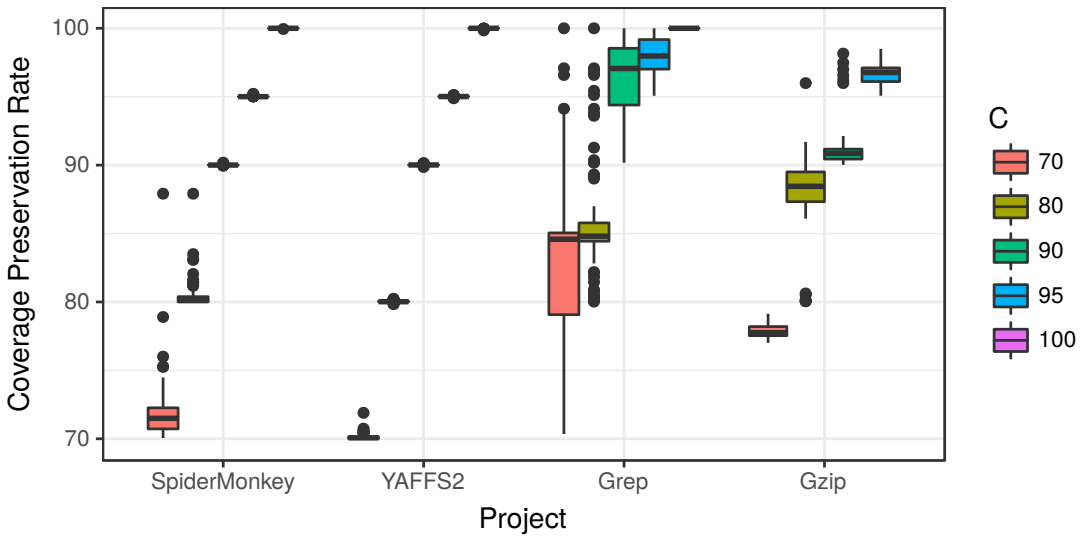

Figure 3.3: CPR for $C\%$-coverage

given $C\%$, but for the other two projects, CPR is sometimes much higher. Overall, the median CPR across different values of C across all projects ranges from 70.07% to 100%.

Figure 3.4 illustrates the relation between different values of N and CPR. The range of median CPR here goes from 78.39% to 100%, which is quite high, showing that preserving even just one mutant leads to CPR close to 80%.

Figures 3.6 and 3.7 summarize the results for MPR on the test cases reduced using $C\%$-coverage and N-mutant, respectively. For $C\%$-coverage reduction, the median MPR ranges from 41.51% to 100% across all projects and all values of C. Concerning general trends, we see that MPR is positively correlated to the value of C: more coverage preserved yields more mutants killed. With C of 95 or higher, the reduced test cases have the median MPR of at least 70% for all projects. Kendall-τ values for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip` were 0.89, 0.80, 0.67, and 0.76, respectively, all with $p < 0.001$, showing a strong positive correlation between the value of C and MPR.

Comparing across the projects, we see that `YAFFS2` has the lowest median MPR when reduced using N-mutant reduction (33.10%). `YAFFS2` test cases are sequences of function calls to the file-system API, such as `mount`, `open`, or `close`. There is little dependency across those functions (e.g., only a few functions call one another), so it is the `YAFFS2` test cases

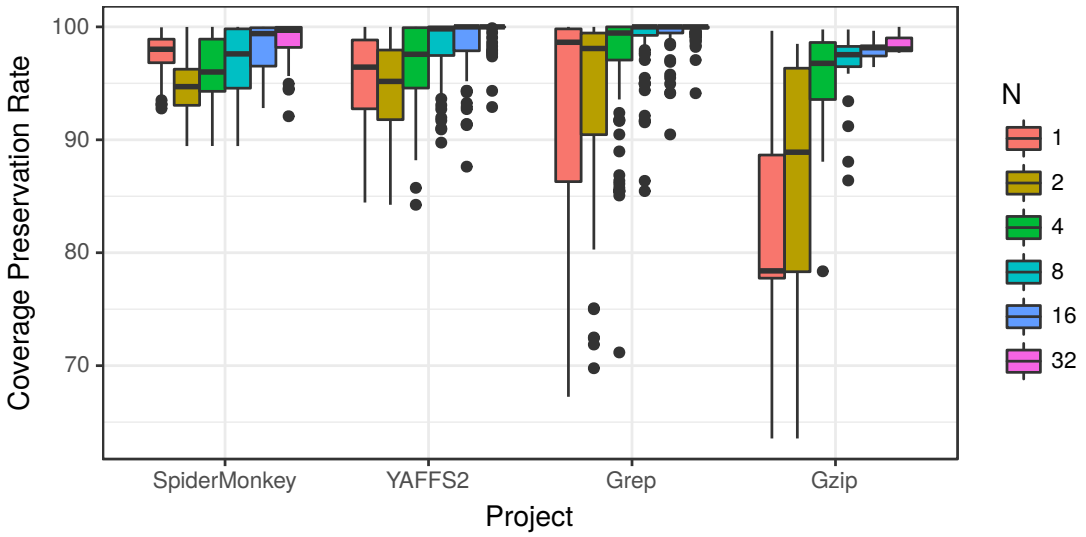

Figure 3.4: CPR for N-mutant

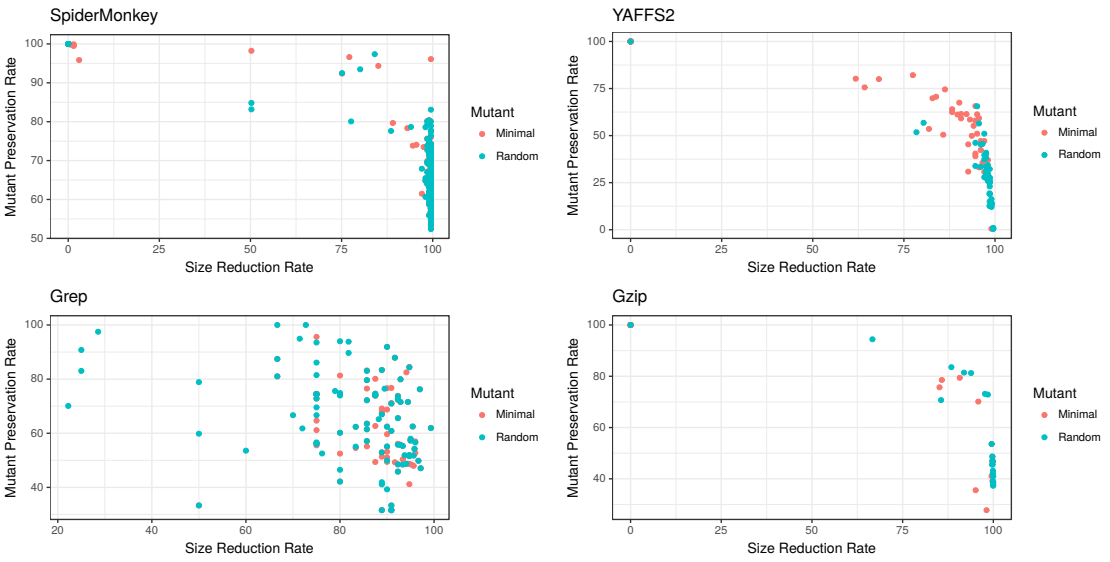

Figure 3.5: SRR vs. MPR, contrasting minimal mutants against randomly chosen mutants, for N-mutant test-case reduction

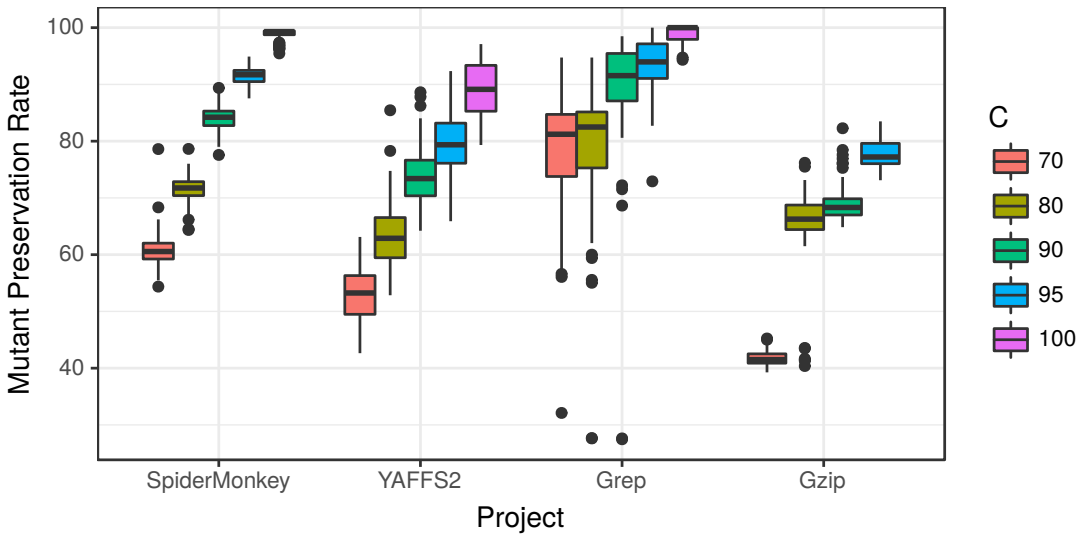

Figure 3.6: MPR for $C\%$-coverage

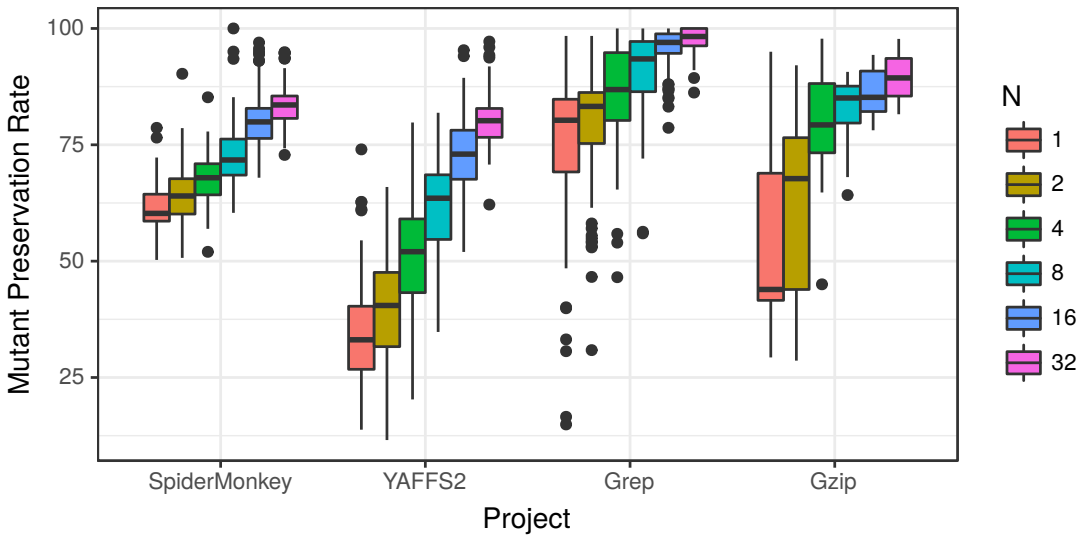

Figure 3.7: MPR for N-mutant

that effectively control the interaction among the functions by the ordering of the API calls. Thus, individual mutants can be isolated reasonably well from the other mutants, due to better decoupling between functions. On the other hand, modules in `SpiderMonkey`, like in any other interpreter or compiler, are deeply intertwined. Thus, each test case exercises multiple functions. As a result, killing a mutant in the parsing module, may also correlate with killing many other mutants in passes before or after parsing, such as lexing or interpretation. Therefore, it is expected that reduced test cases based on even a single mutant in `SpiderMonkey` could still kill a large portion of the mutants killed by the original test cases, with median MPR of 60.26%.

As N grows, MPR of the reduced test cases increases, unsurprisingly: more interdependent mutants can be killed. This observation is validated by the Kendall-τ values: 0.66, 0.69, 0.56, and 0.51 for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip`, respectively, all with $p < 0.001$, suggesting that there is a strong positive correlation between N and MPR. However, a trade-off is that as N increases, the time to perform the reduction increases as well, because intermediate test cases need to be checked against more mutants, and the chance of timeout increases.

In addition to performing N-mutant test-case reduction using N random mutants, we also used minimal mutants. Figure 3.5 shows for each project the relationship between SRR and MPR for test cases from the large, randomly generated test pool reduced using N-mutant with a randomly selected mutant or a minimal mutant. These plots only show the values for $N = 1$, because each test case can kill at most one minimal mutant. Surprisingly, there is no statistically significant difference ($p < 0.001$), except for `YAFFS2`. For `YAFFS2`, test cases reduced based on minimal mutant often result in a better trade-off between SRR and MPR: for the same SRR, test cases tend to have a higher MPR.

3.6.3 RQ3: Trade-Offs

Figure 3.8 shows the trade-off between SRR and CPR for `YAFFS2` test cases reduced using $C\%$-coverage and N-mutant. We show plots only for `YAFFS2` due to space reasons; the plots for the other projects are similar. For $C\%$-coverage, the CPR values cluster very closely with C values, but the SRR values vary, with higher SRR usually corresponding to lower CPR. For N-mutant, many test cases have high SRR, but CPR values vary widely.

Figure 3.9 shows the trade-off between SRR and MPR for `SpiderMonkey` test cases. Again, we show plots only for `SpiderMonkey`; the other projects are similar. For $C\%$-

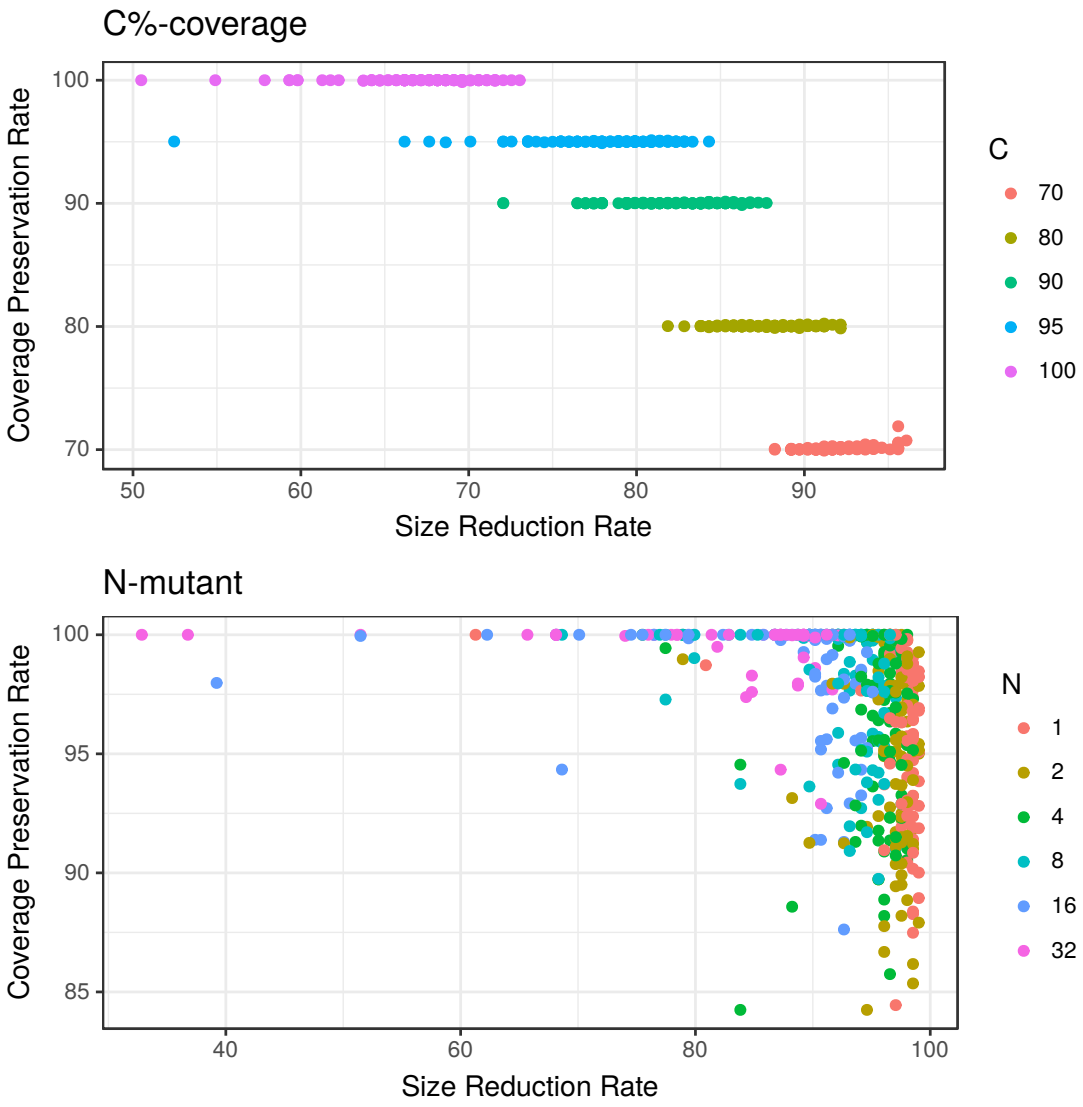

Figure 3.8: SRR vs. CPR for `YAFFS2`

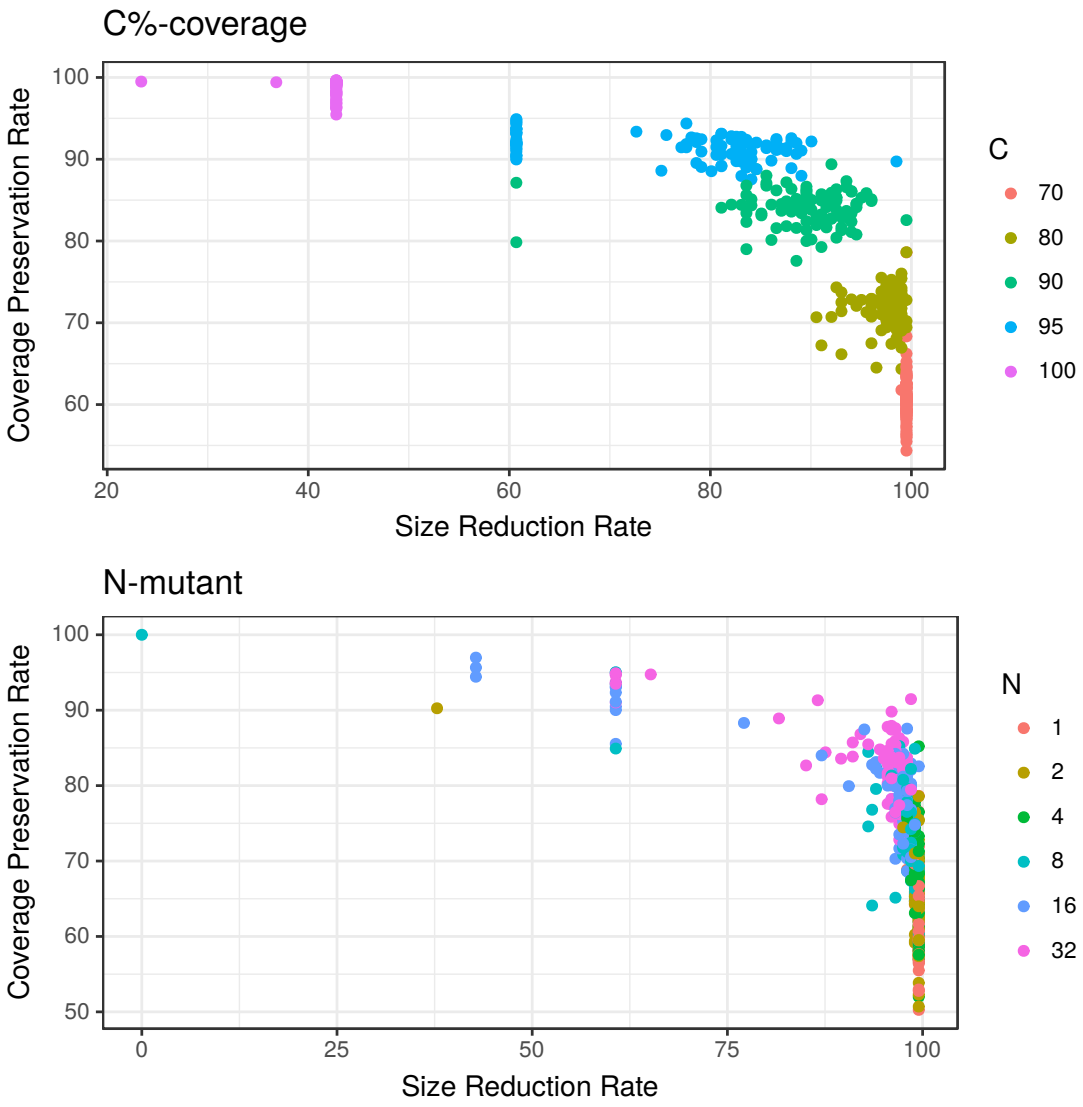

Figure 3.9: SRR vs. MPR for `SpiderMonkey`

coverage, we obtain good SRR and MPR without preserving all coverage: many points for $C = 90$ or $C = 95$ cluster in the upper-right of the plot. For N-mutant, more reduced test cases have high SRR, and larger N values have higher MPR.

Finally, Figure 3.10 visualizes the trade-off between CPR and MPR for all projects. For both plots, we see a linear correlation between CPR and MPR, especially for test cases reduced using $C\%$-coverage. This trend suggests that the more statements a test case covers the more mutants it kills. For N-mutant, especially for `YAFFS2`, there is more clustering towards the right side of the plot, indicating that even with a high CPR, MPR can still vary widely for the test cases reduced using N-mutant.

3.6.4 RQ4: Comparison with Random

We also compared our approaches to simple random test-case reduction that simply forces a certain size reduction on test case. For each test case reduced using non-adequate test-case reduction, we generate three reduced test cases of exactly the same size, by randomly removing parts from the original test case. SRR is exactly the same for a randomly reduced test case as for its corresponding test case. Therefore, we measure only CPR and MPR for these randomly reduced test cases.

Figure 3.11 shows boxplots that compare CPR for test cases reduced using $C\%$-coverage and N-mutant with test cases reduced randomly. We see from these figures that the median CPR computed for test cases reduced by non-adequate test-case reduction is greater than the median CPR computed for the test cases reduced randomly. Figure 3.12 shows the same comparison for MPR. Once again, we see from these plots that the median MPR computed for test cases reduced by non-adequate test-case reduction is greater than the median MPR computed for the test cases reduced randomly. The median CPR/MPR for test cases reduced using non-adequate test-case reduction is greater than the median CPR/MPR for the test cases reduced randomly. A values of randomly reduced test cases are significantly different ($p < 0.001$) from the test cases reduced using non-adequate test-case reduction.

This is hardly surprising, but confirms that our approaches add value. For `YAFFS2`, there is also a specific cause for the extreme differences due to the validity of the reduced test cases: if the original test case is valid, our non-adequate test-case reduction is unlikely to produce an invalid reduced test case. Each valid test case in `YAFFS2` starts by calling a startup function that prepares for mounting the file system. If a test case does not start with this function, the other

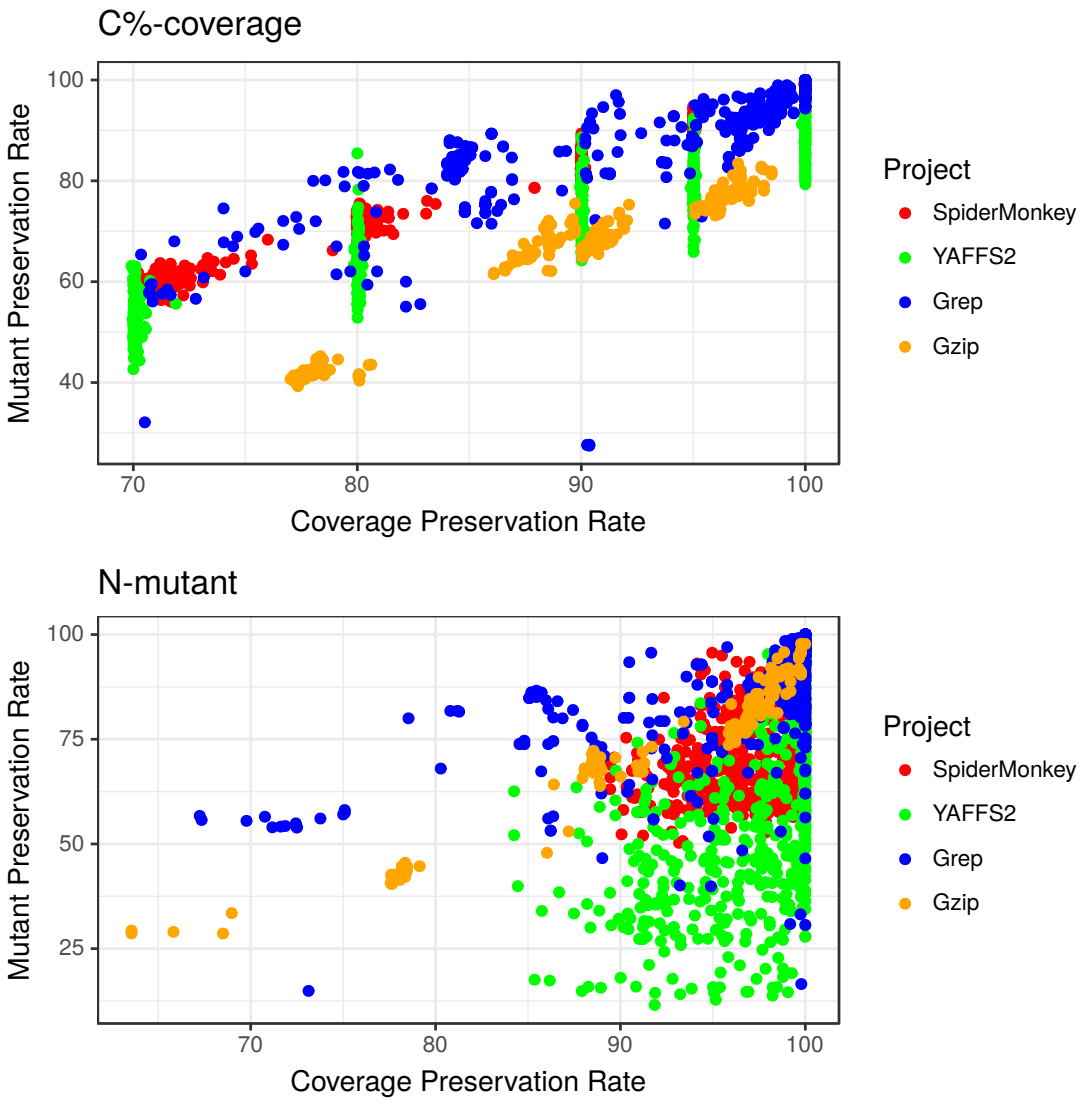

Figure 3.10: CPR vs. MPR for all four projects

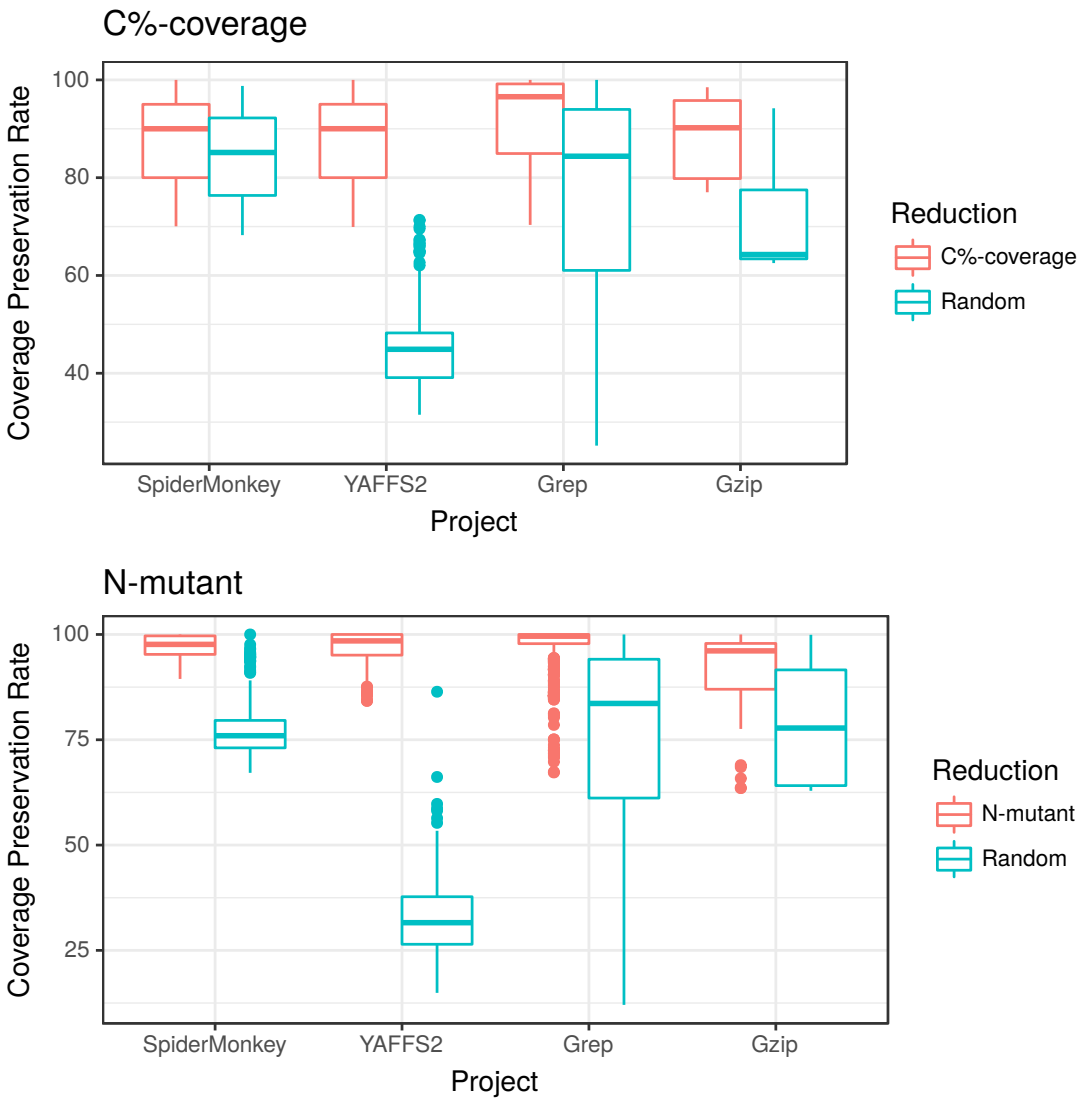

Figure 3.11: Comparing CPR of non-adequate test-case reduction with random test-case reduction

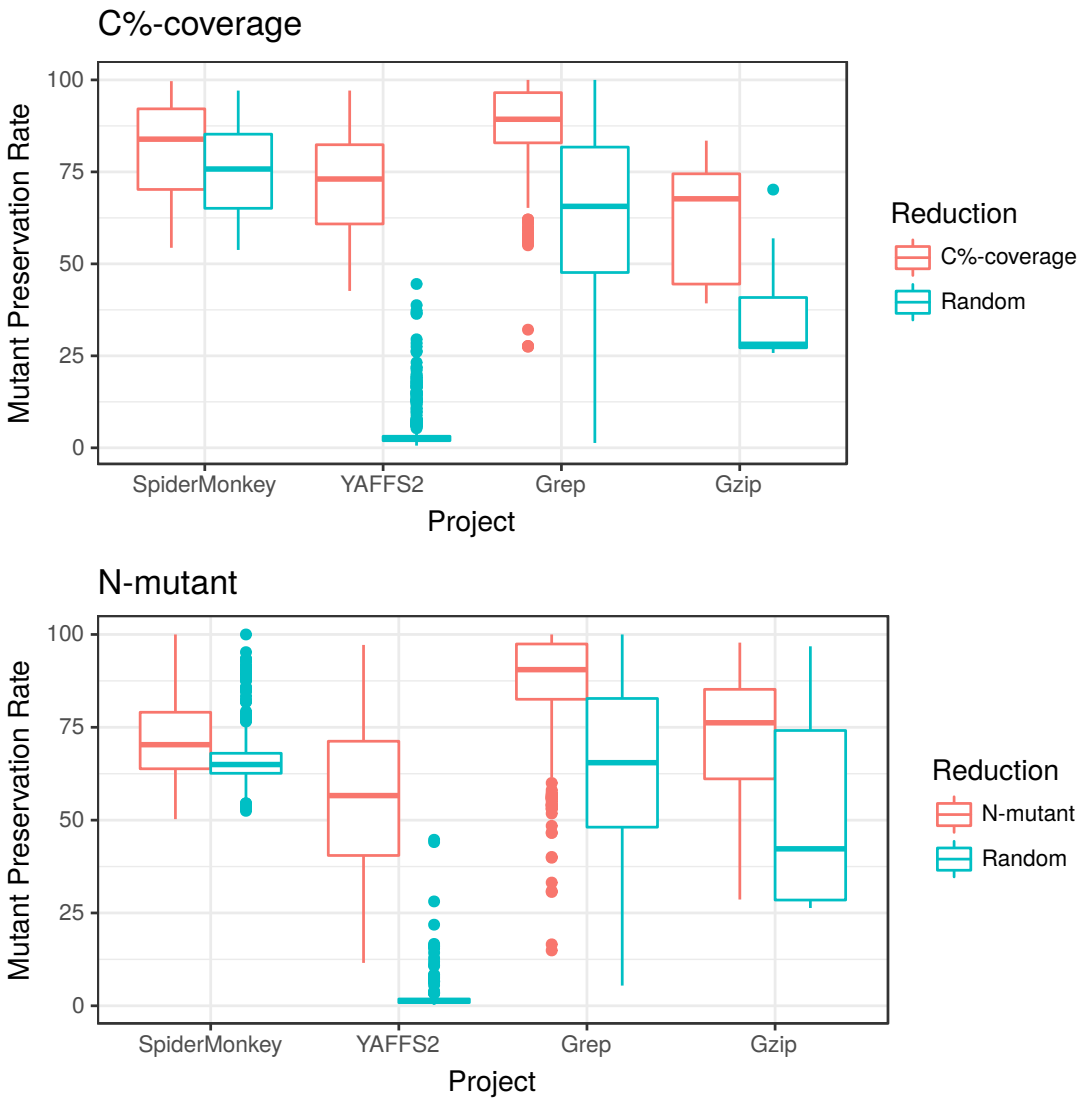

Figure 3.12: Comparing MPR of non-adequate test-case reduction with random test-case reduction

Table 3.2: Time in seconds to perform test reduction

project	$C\%$-coverage			N-mutant		
	Min	Med	Max	Min	Med	Max
`SpiderMonkey`	2	74	1003	1	9	1746
`YAFFS2`	12	102	794	1	24	1700
`Grep`	1	1	15	1	5	483
`Gzip`	4	430	1544	1	82	1799

function calls in the test case fail. The random test-case reduction is unaware of this, so if it has to reduce a sequence of 200 function calls to 4, each function call, including the startup function, has only $\frac{4}{200} = 2\%$ chance to be in the reduced test case, i.e., there is a high chance the reduced test case does not include the startup call and is invalid.

3.7 Discussion

Inter-dependencies among mutants. From the MPR values for N-mutant reduction, we see that focusing the reduced test case to preserve only a small number of mutants killed by the original test case still kills a large fraction of all those mutants. For example, by reducing test cases based on only one mutant (i.e., $N = 1$), the median MPR values are 60.26%, 33.10%, 80.28%, and 43.92% for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip`, respectively. These high MPR values for such a small N suggest that many mutants killed by a test case have strong dependencies. Figure 3.13 illustrates this. The x-axis shows the ratio of N to the total number of mutants killed by the original test case, i.e., $N/Mut(t_o)$, and the y-axis shows the corresponding MPR. For space reasons, we show plots only for `YAFFS2` and `Grep`; the other two projects are similar to `YAFFS2`, but `Grep` is different from all others. We see that a test case reduced based on less than 0.5% of the mutants can still kill more than 50% of originally killed mutants. Note that when a test case reduced to kill some mutant M_1 also kills another mutant M_2, it does not imply that M_1 subsumes M_2 in the sense that *all* tests killing M_1 also kill M_2 [47].

Time for non-adequate test-case reduction. The time for reducing a test case depends on (1) the number of parts in the test case, (2) the time to execute the test case, and (3) the cost of computing coverage or mutants killed. Table 3.2 summarizes the time required for test-case reduction in our experiments. For 50% of the test cases in `SpiderMonkey`, `YAFFS2` and `Grep`, both $C\%$-coverage and N-mutant non-adequate test-case reduction finish relatively

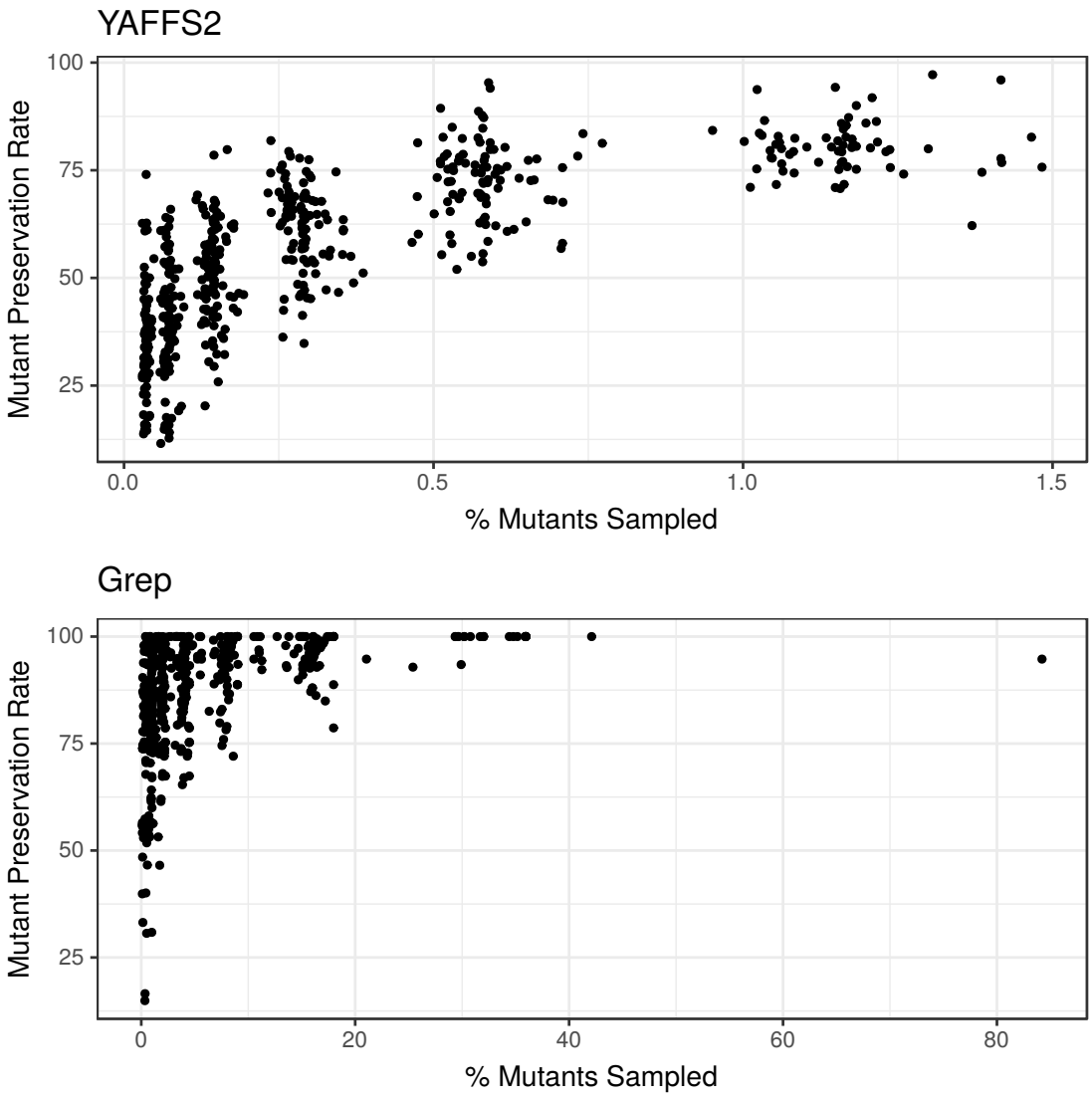

Figure 3.13: Percentage of mutants used for N-mutant test-case reduction vs. MPR

fast (under two minutes for $C\%$-coverage and under one minute for N-mutant). `Gzip` has significantly more parts (up to 3,500) than the other projects, which increases the time needed for reduction. In our experiment, all coverage-adequate reduction (i.e. $C = 100$) of `Gzip` test cases failed due to timeout, but all $C\%$-coverage non-adequate test-case reduction finished within the time limit, with 50% of them being reduced in under ten minutes.

3.8 Threats to Validity

Based on our results, it appears that non-adequate test-case reduction can substantially reduce the size of test cases while still preserving much of test case quality. As usual, experimental result may not generalize to other projects beyond the four we evaluated or even to other test cases and mutants than the ones we used for these projects. A particular threat is how we measure quality. We do not consider some interesting metrics at all (e.g., the execution time of reduced test cases), and the ones used are imperfect.

MPR considers all the mutants killed after performing N-mutant test-case reduction even though the reduction already uses some killed mutants as guidance to reduce the test case; one may argue that by construction the reduced test cases will be good by this metric, or, dually, that this metric is bad. However, we perform non-adequate test-case reduction that does not aim to preserve *all* mutants killed by the original test case, while MPR does consider all mutants killed. Therefore, we do not produce test cases that *necessarily* have a high MPR. Moreover, we also measure the CPR of these reduced test cases, and we do not use coverage to guide N-mutant test-case reduction.

Mutants of C code can introduce undefined behavior. For example, a mutant that removes initialization of a local variable can introduce such behavior. We did not explicitly remove such mutants, but we expect them to be relatively few. Therefore, we generate a large number of mutants for each project, reducing the chance that mutants that introduce undefined behavior significantly bias our results.

Another problem was that of the non-deterministic load on the shared high-performance cluster. Due to nodes having different configurations, and with different loads, a fixed timeout of 30 minutes may not correspond to the same amount of reduction. Hence, some mutants that happened to be evaluated on a slow machine may have been considered killed due to timeout while similarly slow-running mutants evaluated on a fast machine may have managed to complete successfully, thereby not considered killed.

3.9 Related Work

Test-case reduction aims to reduce the size or complexity of test cases while preserving some desirable properties of these test cases. Reduction is essentially a search in the space of possible modifications to the original test case. In many uses, the only modification allowed is removing a part of the test case [55, 57, 63].

One goal of test-case reduction is to speed up testing, and this goal is shared with many techniques for regression testing, including regression test selection, test prioritization, and test-suite reduction [61]. The most similar to test-case reduction is test-suite reduction. Whereas test-case reduction aims to reduce a single test case, test-suite reduction aims to reduce the size of an entire test suite while preserving some desirable properties for the reduced test suite. Many studies investigated test-suite reduction techniques (e.g., [33,35,51]), including our recent work on non-adequate test-suite reduction [54]. However, this paper presents the first study of non-adequate test-case reduction. Test-case reduction and test-suite reduction can be easily combined [28], either in succession or in tandem.

Delta-debugging [63] is the best known technique for reducing the size of a failing test case: it reduces the test case so that it still fails but no single part can be removed without passing. Cause reduction [24,28] generalizes delta-debugging by reducing a test case so that it still has the same coverage (or another property) but no single part can be removed without losing coverage (or another property). Our non-adequate test-case reduction further generalizes cause reduction by not requiring a test case to preserve the complete property the original test case satisfies.

3.10 Conclusion

Having smaller test cases is desirable for developers: such test cases run faster and make debugging easier. Test-case reduction reduces the size of test cases. Previous research has studied how to conduct test-case reduction while completely preserving some property of the original test case, e.g., failure or coverage. We evaluate a more general approach to test-case reduction, called non-adequate test-case reduction, that allows only partially preserving a property. Specifically, we propose and evaluate $C\%$-coverage and N-mutant. Our results show that non-adequate test-case reduction can substantially reduce the size of test cases while still preserving a large percentage of all coverage or mutants killed by the original test cases. For $C\%$-coverage in particular, simply giving up on a very small percentage of coverage can greatly reduce reduction

time and produce a higher gain in size reduction than the associated loss in coverage. The idea of non-adequate test-case reduction greatly expands the options available in exploring trade-offs between test suite size (measured by adding sizes of individual test cases) for fault-detection capability.

3.11 Acknowledgements

We thank Nicholas Lu and Michael Hilton for comments on an earlier draft of this paper. This research was partially supported by the National Science Foundation Grant Nos. CCF-1054876, CCF-1409423, and CCF-1421503. Darko Marinov and August Shi also gratefully acknowledge the Google Faculty Research Award.

Chapter 4: Conclusions and Future Work

The goal of this research was to demonstrate that processing generated tests is profitable, as they can be used for other tasks in software testing. We presented two applcations of generated tests: (1) recommendation of new configuration for generation of focused random tests, and (2) reduction of tests for quick testing scenarios.

In this research we demonstrated with using collected statistics on code coverage of generated tests and swarm testing, it is possible to produce focused random tests — truly random tests that nonetheless target specific source code. While results for the various strategies for directed swarm testing vary, in general the method is able to increase the frequency with which tests cover targeted code by a factor often more than 2x, and sometimes up to 8 or 9x. This approach is readily applicable to existing, industrial-strength random testing tools for critical systems software, and therefore out-of-the-box scalable to applications such as testing production compilers and file systems.

We also evaluated a more general approach to test-case reduction, called non-adequate test-case reduction, that allows only partially preserving a property–in contrast to cause reduction that preserves the property completely. Specifically, we propose and evaluate $C\%$-coverage and N-mutant. Our results show that non-adequate test-case reduction can substantially reduce the size of test cases while still preserving a large percentage of all coverage or mutants killed by the original test cases. For $C\%$-coverage in particular, simply giving up on a very small percentage of coverage can greatly reduce reduction time and produce a higher gain in size reduction than the associated loss in coverage. The idea of non-adequate test-case reduction greatly expands the options available in exploring trade-offs between test suite size (measured by adding sizes of individual test cases) for fault-detection capability.

4.1 Future Work

Leveraging generated tests for different tasks can help us to understand tests better and guide us to improve the quality of tests.

Directed random testing can be extended to regression testing using dominators of changes

as the test targets. Coverage of dominators can increase the probability of covering the changes.

Evaluation of non-adequate test-case reduction, described in Chapter 3, on different subjects shows that N-mutant for a small N preserves a large portion of mutants. This observation can be used to quantify the subsumption relation between mutants and potentially that detection of one can imply that the other one will be detected.

Bibliography

[1] Software fail watch: 2016 in review. https://www.tricentis.com/resource-assets/software-fail-watch-2016/.

[2] Mohammad Amin Alipour and Alex Groce. Bounded model checking and feature omission diversity. In *International Workshop on Constraints in Formal Verification*, 2011.

[3] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, pages 21–30, 2014.

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[5] J. H. Andrews, A. Groce, M. Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *ASE*, pages 19–28, 2008.

[6] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 144–153, New York, NY, USA, 2007. ACM.

[7] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[8] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness. In *International Symposium on Software Testing and Analysis*, pages 265–275, 2011.

[9] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Formal analysis of the effectiveness and predictability of random testing. In *International Symposium on Software Testing and Analysis*, pages 219–230, 2010.

[10] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 559–562, Piscataway, NJ, USA, 2015. IEEE Press.

[11] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, January 2010.

[12] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[13] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, volume 48, pages 197–208. ACM, 2013.

[14] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[15] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*, pages 120–125, 2012.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[17] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using CLP. In *Automated Software Engineering*, pages 482–493, 2015.

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

[19] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.

[20] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA*, pages 302–313, 2013.

[21] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[23] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *ICSE*, pages 72–82, 2014.

[24] A. Groce, M.A. Alipour, Chaoqiang Zhang, Yang Chen, and J. Regehr. Cause reduction for quick testing. In *ICST*, pages 243–252, 2014.

[25] A. Groce, Chaoqiang Zhang, M.A. Alipour, E. Eide, Yang Chen, and J. Regehr. Help, help, I'm being suppressed; The significance of suppressors in software testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 390–399, Nov 2013.

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

[27] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 243–252. IEEE, 2014.

[28] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta debugging, even without bugs. *STVR*, 26(1):40–68, 2015.

[29] Alex Groce, Alan Fern, Jervis Pinto, Tim Bauer, Amin Alipour, Martin Erwig, and Camden Lopez. Lightweight automated testing with adaptation-based programming. In *IEEE International Symposium on Software Reliability Engineering*, pages 161–170, 2012.

[30] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[31] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA, 2012. ACM.

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

[33] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *ICSE*, pages 738–748, 2012.

[34] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[35] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *TOSEM*, 2(3):270–285, 1993.

[36] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[37] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE*, pages 435–445, 2014.

[38] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Combining stochastic grammars and genetic programming for coverage testing at the system level. In *Search-Based Software Engineering*, pages 138–152, 2014.

[39] Sunghun Kim, E.J. Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, March 2008.

[40] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[42] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.

[43] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, 2013.

[44] P. McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, March 2011.

[45] E. Nagai, A. Hashimoto, and N. Ishura. Scaling up size and number of expressions in random testing of arithmetic optimization in c compilers. In *Workshop on Synthesis and System Integration of Mixed Information Technologies*, pages 88–93, 2013.

[46] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *ISSTA*, pages 354–365, 2016.

[48] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, pages 936–946, 2015.

[49] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[50] Edward L. Robertson and Catharine M. Wyss. Optimal tuple merge in NP-complete. Technical Report TR599, Indiana University Bloomington, July 2004.

[51] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *STVR*, 12(4):219–249, 2002.

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

[53] Jesse Ruderman. Releasing jsfunfuzz and DOMFuzz. `https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/`, 2015.

[54] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014.

[55] Donald R. Slutz. Massive stochastic testing of SQL. In *VLDB*, pages 618–622, 1998.

[56] Dmitry Vyukov. gosmith: Random Go program generator. `https://code.google.com/p/gosmith/`.

[57] David B. Whalley. Automatic isolation of compiler errors. *TOPLAS*, 16(5):1648–1659, 1994.

[58] E. B. Wilson. Probable inference, the law of succession, and statistical inference. *J. of the American Statistical Assoc.*, 22:209–212, 1927.

[59] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.

[60] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[61] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[62] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA*, pages 140–150, 2007.

[63] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2):183–200, 2002.

[64] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *ISSTA*, pages 160–170, 2014.