ABSTRACT

Tests generated by automated test generators are seldom processed for further use in software testing. The goal of this research is to find techniques for processing such generated tests. These techniques should produce new test cases by transforming generated tests, or use information in the tests to guide future test generation.

The first part of this dissertation intends to devise a technique to use the information in the generated tests to improve test generation. The proposed technique would improve the frequency of coverage for important code targets in swarm testing. We evaluate the effectiveness of this technique by comparing the frequency of coverage in (traditional) un-directed swarm testing and directed swarm testing.

In the second part of this dissertation, we propose a set of test reduction criteria to generate new test cases by reducing already generated test cases. The goal of the proposed reduction criteria is to speed up testing while preserving (most of) fault-detection capability of the original tests. Effectiveness of the reduction techniques is measured by comparing the size, coverage and mutation-killing power of the reduced tests with the original tests.

1. INTRODUCTION

Automatically generated tests have been successful in uncovering new bugs in many important programs and libraries. As the cost of generating new test cases decreases, tendency to discard already generated tests increases. For example, in most fuzzing scenarios, the tests that do not find bugs are discarded.

Generated tests, in particular randomly generated tests, have rarely been subject to analysis and transformation to create new tests or improve the test generators. Our vision is that generated tests, like any other software artifacts, contain useful information about the software under test. This information can be utilized in generating new tests, or improving the test generation. In this research, we introduce focused testing and non-adequate reduction which use test cases and information in the tests to improve the test generation and generate new tests.

Focused Testing

Random testing [22] (sometimes called fuzzing) is now widely recognized as an effective approach for testing software systems, including compilers [38,39,46], standard libraries [34], static analysis systems [9], and file systems [19]. Random testing is used in both complex custom-built testing systems 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 [5]. 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 [50]. It is seldom practical to implement different random testers for all the potential focuses that might be needed, and the most powerful random testers [19,38,46] tend to be based on generating complete inputs (e.g. programs or sequences of file system calls) as whole system tests for the Software Under Test (SUT), and do not even attempt to provide module-level testing. Of course, tests generated by a random tester can be selected based on what they happen to have covered by accident, but replaying stored pre-existing tests defeats much of the point of random testing.

Techniques for making better use of random tests in situations requiring more focus, such as regression testing, are now appearing [16], 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 [30]). 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 case in which 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 [13, 45] and search-based techniques [24, 33]) 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. Hand-tuned whole-program random testers, however, are a popular technique for testing such systems, including C compilers [32, 46], JavaScript engines [26, 38, 39], and Google’s Go language [42]. 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 effectively 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.

**Test Reduction**

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 test suites [15]. Smaller, simpler test cases are also easier to understand and enable effective debugging, which provided the original motivation for delta-debugging [49], a technique for reducing the size of failing test cases. The advantages of smaller test cases result in random test generation being often combined with delta-debugging, and the development of effective reduction techniques is itself an important research area [23, 25, 37, 40, 48].

Finally, smaller test cases make it possible for test case selection and prioritization to operate at a much finer granularity than when applied to test suites mostly consisting of large, complex test cases.

However, it is also true that smaller test cases, all else being equal, detect fewer faults than larger test cases [7, 19]. The trade-off between size and effectiveness for individual test cases is similar to the trade-off between smaller and larger test suites which are sets of individual test cases. Researchers have extensively studied test-suite reduction [23, 25, 37, 40, 48], 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, many of which are summarized in a survey by Yoo and Harman [48]. 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). Most of these 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. However, recent work on non-adequate test-suite reduction [40] evaluated only partially preserving the property of interest, e.g., if that property is code coverage, keeping 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 requires “slicing and dicing” the atomic parts that make the test case. For example, if a test case is a unit test comprising 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 and depends 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. While test-suite reduction has been studied in depth at least since 1993 [25], test-case reduction research is much more recent.

Zeller and Hildebrandt proposed delta-debugging [49], the best known test-case reduction technique, usually applied to reduce a failing test case to a minimal test case that still fails. Recently, Groce et al. proposed cause reduction [15, 17] as a generalization of delta-debugging, and used it to reduce a (passing or failing) test case, while preserving coverage obtained by the original test case. That work uses cause reduction to completely preserve coverage: the reduced test case has exactly the same coverage as the original test case.

We refer to such reductions as 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, but the original test case itself may provide far from adequate code coverage.

The utility of non-adequate reduction for test suites [40] naturally suggests that non-adequate reduction may be useful for test cases as well. Non-adequate reduction for 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 reduction can be obtained (some test cases cannot be reduced substantially without sacrificing at least some coverage or mutation detection), and increases the time required to reduce test cases (because searching for an adequate reduction is more difficult than finding a reduction with a similar, but not the same, quality).

Combining the recent ideas of non-adequate reduction for test suites [40] and (adequate) reduction for test cases [15, 17], this research empirically evaluates a new combination: non-adequate reduction of test cases. To the best of our knowledge, ours is the first such evaluation.

**Contributions**

The contributions of this dissertation would be:

- It introduces new goal of increasing the frequency of coverage for focused testing.
- It introduces a method for for generating focused random tests that uses the information in test cases of test suites to generate new test suites that with high frequency of coverage of targets.
- It generalizes previous work in test case reduction.
It evaluates different test reduction techniques to generate new tests from existing tests.

**Organization.** First, we explain directed swarm testing method for focused random test in Section 2. Section 3 describes non-adequate reduction techniques. Project’s to-do list and timeline of the projects are described in Section 4.

2. DIRECTED SWARM TESTING

In this study, we propose a method, directed swarm testing, that makes the generation of random tests that focus on selected code targets possible. Using swarm testing [21], a variation of random testing, and recording statistical data on past testing results [20] 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 [46] and concerned about the correctness of a particular set of seldom-executed lines in the implementation of a complex optimization 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 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 that 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.

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 code that warrants 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-tuned optimized random testing) and modified test generators using swarm testing.

Contributions of this part of dissertation 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.
- 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).
- Empirical results of effectiveness of these strategies on finding real faults in a large software system.

2.1 Preliminary

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.

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 [21].

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 [20], 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 [44] 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 true 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 $t$:

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

$F$ 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 [20].

Comparing the accuracy of Wilson score and other measures.

2.2 Related Work

The most closely related work is the work of Groce et al. introducing swarm testing [21] and the notions of triggers and suppressors [20]. We expand on that work by using the concepts introduced to enable a practical way to generate focused random tests: diverse random tests, with the advantages of random testing, but with a higher coverage frequency for targeted code.

There are several approaches for generating a test case that covers a chosen source code target once. Of these, search-based testing [24,33] and (dynamic) symbolic execution [13,45] are the most notable ones. Symbolic execution [31] 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 [13]. Search-based testing reduces the problem of covering a particular entity in the program into a search problem and uses search techniques, such as genetic algorithms and hill climbing, to solve this problem [24,33].

There are many previous efforts to improve the test cases generated by random testing. Randoop [34] generates tests for object-oriented programs by calling random APIs, but uses feedback to guide test sequence creation. Nighthawk [3] 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 [6,7] aims to improve random testing by using a distance measure to select more uniformly distributed tests, though its actual effectiveness in practice has been criticized [4]. ABP-based testing uses reinforcement learning to guide test generation [18].

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 [50]. Symbolic execution often simply fails to scale to very large systems with complex input. The approach proposed here 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.

2.3 Technique

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 $T$ at once, as is often useful in testing changed code, it may only target some subset of $T$ in each individual test generation. A directed swarm testing strategy is effective if it increases the average rate at which tests hitting targets 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, 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. However, it is easy to construct a set of targets for directed swarm testing in this situation: the four 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). 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 he 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 compute triggers and suppressors 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 [20]. Moreover, the cost of re-baselining is low enough that there is little reason not to routinely collect new trigger and suppressor information, especially as this is a by-product of an independently useful effort (checking code coverage).

2.4 Configuration Strategies

Figure 1 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 empirical work of Groce et al., [20], be used to compute, for each source code target \( t \) (in this our 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 \( T \). 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; the causal patterns may not operate in the anticipated way when using a very different configuration distribution, due to the combinatoric complexities of hand-tuned random testers. 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.

2.5 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 \( C_i = \{f | f \in T(t) \cup \text{randomSample}(f | f \notin S(t))\} \), where \( \text{randomSample} \) 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 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

³A common statement dominated by all targets can also be used, if such a statement exists.
addition to the basic empirical question of effectiveness, the idiosyncrasies 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.

2.6 Multiple-Target Strategies

For multiple targets, \( T \), our strategies reduce the problem to that for single targets \( t \in T \):

1. **Round-robin**: The Round-robin strategy simply applies a single-target strategy in a round-robin fashion, for \( t \in T \).

2. **Merging**: The Merging strategy attempts to merge triggers and suppressors for targets in \( T \) to produce a minimal set of targets (each of which may represent multiple real 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 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 \( T' \) for targets \( T \). Given targets \( t_i, t_j \in T \), we say \( t_j \) *subsumes* \( t_i \), denoted \( t_j \sqsubseteq t_i \), if and only if, \( S(t_i) \subseteq S(t_j) \land T(t_i) \subseteq T(t_j) \). 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.

1: for \( \forall t_i \in T \) do
2: if \( \exists t_j \in T \mid t_j \sqsubseteq t_i \) then
3: \( D = D \cup t_i \)
4: end if
5: end for
6: return \( t \in T \mid t \notin D \)

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 when neither is stricter than the other, treating them as one target \( t' \), with \( T(t') = \{ f \mid f \in T(t_1) \lor f \in T(t_2) \} \), \( S(t') = \{ f \in S(t_1) \lor f \in S(t_2) \} \), and \( I(t') = \{ f \notin S(t') \land f \notin T(t') \} \). 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 [36]. We implemented an SMT-based exact solver for merging targets using Z3 [10], 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 [20], 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.

### 2.7 Evaluation Methodology

We used three medium-moderately large C programs (shown in Table 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 test generation tools descended from those used to test the file systems for NASA’s Curiosity Mars Rover [19], and applied in previous work on combining random testing and symbolic execution [50]. For GCC, we used the Csmith [46] 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 [38], 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 [46] 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 complex 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 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 triggers/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 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.

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.

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 $\frac{HF_u}{HF_d}$ to measure

---

### Table 1: Experimental Subjects

<table>
<thead>
<tr>
<th>SUT</th>
<th>LOC</th>
<th>Fuzzer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAFFS2</td>
<td>1B</td>
<td>yaffs2tester</td>
<td>Flash File System</td>
</tr>
<tr>
<td>GCC 4.7</td>
<td>800K</td>
<td>Csmith</td>
<td>LLVM and C++ Compiler</td>
</tr>
<tr>
<td>SpiderMonkey</td>
<td>118K</td>
<td>jsfunfuzz</td>
<td>JavaScript Engine For Mozilla</td>
</tr>
</tbody>
</table>

### Table 2: Experimental Parameters

<table>
<thead>
<tr>
<th>SUT</th>
<th># Features</th>
<th>Seed time (min.)</th>
<th>Directed time (min.)</th>
<th># Undirected Suites</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAFFS2</td>
<td>43</td>
<td>18</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>GCC 4.4.7</td>
<td>25</td>
<td>60</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>SpiderMonkey</td>
<td>171</td>
<td>30</td>
<td>10</td>
<td>54</td>
</tr>
</tbody>
</table>

---

\(^3\)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.
the effectiveness of directed testing in hitting targets more frequently. Note that directed test suites with $HF_d > 1.0$ offer improvement over undirected test suites. This is the measure a developer wants to increase via targeting.

2.8 Research Questions

Our experimental results address the following six basic research questions. In this proposal, we only include the answer to the first research question:

- **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 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 testing is the only directed testing approach we evaluate (and, to our knowledge, the only one applicable to our subject programs).

Table 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, “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.

**RQ1 and RQ2: Single-Target Strategies**

Table 3 shows the results for single-target directed swarm testing under different directed testing strategies, including $p$-values for Wilcoxon tests. Table 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 hitting 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.
3. TEST REDUCTION

We propose two techniques for non-adequate test-case reduction; to the best of our knowledge, ours are the first such techniques. Our techniques reduce a larger test case into a smaller test case while only partially preserving some property. Specifically, the C%-coverage technique reduces an original test case while preserving at least C% of the test case’s original code coverage, and the N-mutant technique reduces an original test case while preserving that the reduced test case kills at least N mutants randomly selected from the set of all mutants killed by the original test case. We refer to these techniques as “non-adequate” because they do not necessarily preserve completely either the code coverage or all mutants killed. However, the reduced test cases could in theory cover code elements or kill mutants that the original test cases do not, even if the reduced test cases do not cover all code elements or kill all mutants that the original test cases do; in fact, the reduced test cases could even cover more code or kill more mutants overall.

Effectively, our proposed non-adequate test-case reduction generalizes the previously proposed test-case reductions. By parameterizing the level to which a property needs to be preserved in the reduced test case, we allow more freedom to explore trade-offs between (bigger) size reductions and (smaller) preservation of the fault-detection capability [40, 47]. For example, cause reduction becomes just a special case of our C%-coverage technique when we set C = 100, and preserving to kill only one mutant that would properly encode one failure can mimic delta debugging with and preserving to kill only one mutant that would properly preserve at least 100% of the test cases, and for each computing coverage of all modules, scanning all the coverage files, and comparing them with the coverage of the original test case at each reduction step. For example, Groce et al. [15] reported that cause reduction of a large test case for the GCC compiler could take days. Moreover, preserving 100% of the coverage may not be necessary, because a test case that preserves less may still have high quality.

C%-coverage relaxes the coverage requirement; the reduced test case need not necessarily preserve 100% but should preserve at least C% of the statements covered by t_o. (We note again that t_o itself may have a very high or very low quality with respect to the entire program under test.) 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 “I-minimal” test cases where no single part of the test can be removed without violating some reduction requirements.

3.1 C%-coverage Reduction

Suppose that we are given a test case t_o that covers some set Cov(t_o) of statements in the program under test. Previously proposed cause reduction [15] produces a reduced test case that still preserves the coverage of the entire set (but could potentially cover even more statements). However, cause reduction of large test cases for complex software can be highly inefficient, because it involves searching many test cases, and for each computing coverage of all modules, scanning all the coverage files, and comparing them with the coverage of the original test case at each reduction step. For example, Groce et al. [15] reported that cause reduction of a large test case for the GCC compiler could take days. Moreover, preserving 100% of the coverage may not be necessary, because a test case that preserves less may still have high quality.

C%-coverage relaxes the coverage requirement; the reduced test case need not necessarily preserve 100% but should preserve at least C% of the statements covered by t_o.

**Definition 1.** C%-coverage test-case reduction produces a reduced test case t_r that covers at least C% of the statements that are 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 defined with respect to the original test case and not with respect to the entire program under test, i.e., we do not want simply \(|Cov(t_r)|/|Cov(t_o)| \geq C\%\), because \(t_r\) could then be covering statements not related to \(t_o\). Viewed this way, cause reduction can be defined as requiring \(|Cov(t_r) \cap Cov(t_o)|/|Cov(t_o)| = 100\%\), or equivalently \(Cov(t_r) \supseteq Cov(t_o)\). In other words, C%-coverage does not put any (direct) requirement for \(|Cov(t_r)|\) and \(|Cov(t_o)|\), so it may be even the case that \(|Cov(t_r)| > |Cov(t_o)|\) if \(t_r\) covers some statements that \(t_o\) does not cover. In other words, C%-coverage does not impose any requirement on the statements that are not covered by the original test case; the reduced test case may or may not cover those statements.

3.2 N-mutant Reduction

We define N-mutant reduction similarly to C%-coverage reduction, but it has three important differences: (1) using mutants instead of statements, (2) preserving an absolute number of selected mutants \(N\) rather than a relative ratio of statements \(C\%), and (3) keeping the set of selected mutants constant throughout all reduction steps rather than allowing the set of mutants to change among reduction steps (as motivated further at the end of this section). Suppose that we are given a test case \(t_o\) that kills some mutants \(Mut(t_o)\) in

\[\text{While we will present and evaluate our C%-coverage technique only for statement coverage, it generalizes to any other coverage, e.g., branch coverage.}\]
the program under test. We could require that the reduced test case $t_r$ kills all those mutants. However, searching for such reduced test case would be extremely inefficient because each step of reduction would require to check that the intermediate test cases kill all the mutants. Also, it is likely unnecessary to preserve all the mutants; the developers may be interested in one specific mutant or a handful of mutants in the proximity of the target mutant. Reducing the test case to preserve a small set of mutants can still produce a highly useful smaller test case, and reducing test cases based on a limited number of mutants is definitely much more efficient than reducing with respect to the entire set of mutants.

$N$-mutant requires the reduced test case to preserve some specific $N$ mutants selected from the set of all mutants killed by the original test case:

**Definition 2.** A $N$-mutant test-case reduction produces a reduced test case $t_r$ that has to kill a specific set of $N$ mutants selected from the set of $\text{Mut}(t_o)$, where typically $N \ll |\text{Mut}(t_o)|$, and the $N$ mutants are selected randomly.

As for $C\%$-coverage, note that the preservation is defined with respect to the original test case and not with respect to the entire program under test, i.e., $\text{Mut}(t_o)$ has to include the $N$ mutants selected from $\text{Mut}(t_o)$, but $N$-mutant does not put any (direct) requirement for $|\text{Mut}(t_o)|$ and $|\text{Mut}(t_o)|$, so it may be the case that $|\text{Mut}(t_o)| > |\text{Mut}(t_r)|$ if $t_o$ kills some mutants that $t_r$ does not kill. In other words, $N$-mutant does not impose any requirement on the mutants not among the $N$ selected mutants (be those mutants killed by the original test case or even not killed by it but only generated): the reduced test case may or may not kill any of those other mutants.

Unlike for $C\%$-coverage, which does not keep the set of statements constant among reduction steps but only requires that a certain number of those statements be covered, $N$-mutant does keep the set of $N$ selected mutants constant, as listed in the point (3) at the start of this section. We did initially experiment with a technique that allowed the set of mutants to change, while requiring only that the number of mutants be preserved through reduction steps be at least $N$. However, our experiments showed that this reduction technique with changing mutants had two undesired effects. First, it produced larger reduced test cases than keeping the set of $N$ mutants constant. Second, it was less efficient than keeping the set of $N$ mutants constant. So paradoxically, not only did the search take longer but also it produced worse results. Both of these reasons are caused by the reduction forcing the search to backtrack many times when an intermediate test case in one step does not kill a mutant that the next step wants to check.

### 3.3 Related Work

Test-case reduction aims to reduce the size or complexity of test cases while preserving particular properties of these test cases. This is essentially a search in the space of possible modifications to the original test case. In most cases, the only modification allowed to the test case is removing or replacing a part of the test case [41, 43, 49].

As the goal of test-case reduction is to speed up testing, it is similar to the many techniques studied in previous projects to speed up regression testing, including regression test selection, test prioritization, and test-suite reduction [48]. The most similar of these techniques 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 measure of quality (typically by some metric of fault-detection capability) for the reduced test suite. Many studies in the past have investigated different test-suite reduction algorithms [23, 25] or how effective test-suite reduction techniques are [37, 40]. We imagine test-case reduction and test-suite reduction can be easily combined, where one can first reduce a test suite to remove any redundant test cases, and then from the remaining test case in the reduced test suite one can perform test-case reduction to speed up the testing process even more.

Delta debugging [49] is the most common technique for reducing the size of test cases. Given a property of interest and a test case, the algorithm reduces the test case to one that preserves the property, such that no single part can be removed without losing the property. Cause reduction [15] is a generalization of delta-debugging, where a test case is reduced until removing any one part causes the reduced test case to not cover everything the original test case covers. Our non-adequate test-case reduction techniques are a further generalization of cause reduction, where we relax the constraint of needing to cover everything the original test case covers and instead accepts a reduced test case that covers at least some percentage.

Mutants of a program are variants of the program with a small syntactic change from the original program. Studies suggest that mutants can be used as proxies for real software faults [2, 28], so if a test suite or an individual test case can kill more mutants, it can potentially detect more real bugs. Moreover, mutants subsume a large number of structural coverage metrics. Thus, it is reasonable to reduce test cases with respect to mutants.

#### 3.3.1 Metrics

In this section, we describe three metrics for evaluating the effectiveness of test-case reduction: Size Reduction Rate (SRR), Coverage Preservation Rate (CPR), and Mutation Preservation Rate (MPR). We make all metrics such that the higher values are better and that the values are normalized to range between 0% and 100%. We also relate the metrics with non-adequate test-case reduction.

**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 $\text{Size}(t)$ 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) measures the reduction in size of the reduced test case relative to the original test case:

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

A higher ratio for SRR is desirable because it indicates that more parts have been removed from the test case, resulting in a smaller reduced test case.
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 is commonly used as a proxy for fault-detection capability to evaluate the quality of test cases; the more code a test case covers the higher chance it can detect a fault, and conversely if a test case fails to cover some part of code there is no way it can detect any faults in that part of the code.

A number of studies have shown that code coverage is a strong indicator of fault-detection capability [11, 14]. We note that while the study by Inozemtseva et al. [27] seems to suggest otherwise, Inozemtseva et al. looked at the effect of coverage after discounting the effect of the size of test suite. If the effect of test suite size is not controlled for, Inozemtseva et al. [27] found a strong correlation between code coverage and fault-detection capability of a test suite.

We use statement coverage as the structural code coverage metric to evaluate quality. Recall that $\text{Cov}(t)$ 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 of the number of statements covered by the reduced test case that are also covered by the original test case, to the number of statements covered by the original test case:

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

A higher ratio for 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 originally covered statements.

Mutation Preservation Rate (MPR)

Another metric commonly used to evaluate the quality of test cases is the number of killed mutants. We measure how effective the reduced test case is at killing mutants when compared to the original test case. Recall that $\text{Mut}(t)$ denotes the set of mutants killed by a test case $t$.

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

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

A higher MPR is desirable because it indicates the reduced test case is better at preserving the ability to kill mutants that the original test case kills. Note that a reduced test case can potentially kill more mutants than the original test case, but CPR does not consider the other mutants that are killed by the reduced test case but not killed by the original test case. Like CPR, MPR can never go over 100%.

3.4 Relating Requirements for Reduction and Metrics

Both the reduction algorithms and the metrics are based on coverage and mutants, but 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 different reductions. For C%-coverage reduction, we know that CPR will be at least C%, but it could be much higher (though always less than or equal 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 $N/|\text{Mut}(t_o)|$, but it could be much higher (in fact, our experiments find that even when $N/|\text{Mut}(t_o)| < 1$, MPR can be quite high), and CPR could again in theory range from literally 0 to 100%.

3.5 Evaluation

This section describes the experiments we designed to evaluate the effectiveness of our C%-coverage and N-mutant non-adequate test-case reduction. Section 3.5.1 describes the projects used in our evaluation, along with some characteristics of their test cases and the mutants. Section 3.5.5 describes our experimental setup.

We ran all our experiments on a high-performance computing cluster composed of commodity computing nodes. Each node housed between 6 to 12 2.6GHz Intel Xeon cores. All experiments together would take several weeks on a single core, mostly due to the high cost of mutation testing.

3.5.1 Projects

Table 4 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 the smallest part of each test case is, the total number of mutants used, and the minimum and maximum number of mutants killed by each test case. We use four small- to medium-size C projects in our evaluation: SpiderMonkey is Mozilla’s JavaScript engine, YAFFS2 is a flash file system that was used in the early Android platforms, and Gzip is the standard Unix utility for compressing/decompressing files.

3.5.2 Test Cases

We use randomly generated test cases for SpiderMonkey, YAFFS2, and Gzip. More precisely, for SpiderMonkey, YAFFS2, and Gzip, we randomly generate test cases for each project, as done in similar previous work [15, 17]. The SpiderMonkey test cases are JavaScript programs randomly generated using a highly successful jsfunfuzz [38] 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 [8, 15, 17]. The Gzip test cases are files that are composed of sequences of 500 to 3,500 random bytes.

Measuring the size of a test case is an open question in the software testing community. Researchers use a variety of metrics to assess size, such as the number of API calls, execution time, number of assertions, etc. Following Groce et al. [15], we define size as the number of the atomic parts of a test case that can be removed while performing test-case reduction. The concrete part differs from one project to another. For SpiderMonkey, a part is a JavaScript statement in the generated program. For YAFFS2, a part is one API call in the generated sequence of API calls. Table 4 summarizes what we define as a part for each project.

Time complexity of test-case reduction building on delta-debugging is quadratic in the number of parts in a test case.
Table 4: Four projects used in our evaluation and some statistics of their test cases and mutants

<table>
<thead>
<tr>
<th>Project</th>
<th>NLOC</th>
<th># test cases</th>
<th>part</th>
<th># mutants</th>
<th>min killed</th>
<th>max killed</th>
<th>test pool</th>
<th># minimal mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderMonkey</td>
<td>81,920</td>
<td>99</td>
<td>A statement of JavaScript program</td>
<td>69,067</td>
<td>8101</td>
<td>12825</td>
<td>890</td>
<td>256</td>
</tr>
<tr>
<td>YAFFS2</td>
<td>10,356</td>
<td>99</td>
<td>One API call</td>
<td>15,046</td>
<td>2071</td>
<td>3413</td>
<td>1000</td>
<td>57</td>
</tr>
<tr>
<td>Gzip</td>
<td>5,129</td>
<td>77</td>
<td>A byte in the input file</td>
<td>7,115</td>
<td>1813</td>
<td>2046</td>
<td>1000</td>
<td>37</td>
</tr>
</tbody>
</table>

For SpiderMonkey, YAFFS2, and Gzip, for which we generate the test cases, we control the number of parts such that test-case reduction can finish in reasonable time. We obtained the specific limits from the initial experiments, trying to finish most test-case reductions within 30 minutes. In particular, we limit each SpiderMonkey test case to be a program consisting of exactly 200 JavaScript statements, 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.

3.5.3 Mutants

We use a mutation-testing tool for C programs developed by Andrews et al. [2] and used in many previous studies. Quoting [2], the tool provides the following four classes of mutation operators:

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

The tool performs source-to-source mutations, so the mutants are source code. We compile each mutant with the highest optimization available in the GCC compiler (-O3) and compare each binary with the others to keep only one representative from each equivalence class of mutants that are trivially compiler-equivalent [35]. (This analysis finds about 15% of the generated mutants to be equivalent.) Table 4 shows the number of mutants we used 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 std::cout, std::cerr, and artifacts produced) differs from the output of the original program.

3.5.4 Minimal Mutants

To evaluate $N$-mutant reduction, we use two methods to select mutants. The first method is random sampling of $N$ mutants from the set of mutants killed by the test case. The second method uses the minimal mutant set, defined by Ammann et al. [1] as the smallest set of mutants that represents the complete set of mutants in terms of faults detected by a given test suite. A minimal mutant set is important in that it can represent all the faults in a set of mutants killed by a test suite, such that killing all mutants in the minimal mutant set guarantees killing all mutants in the full set of mutants.

To define the minimal mutant set, the first step is to construct a minimal test suite from the original test suite. A minimal test suite is a subset of the original test suite that is mutation adequate against the set of mutants killed by the original test suite (i.e., the minimal test suite scores 100% against the mutants killed by the original test suite), and removing any test case from the minimal test suite results in a drop in mutation score. 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 implies killing all the mutants from the original mutant set.

Our strategy was as follows. First, we generated a large test pool of random test cases to represent a test suite for each project. Then, we applied the mutation operators on the projects and obtained the complete set of mutants killed by the test suite. For each test pool and its mutant set, we minimized the test suite with respect to the killed mutant set to obtain the minimal test suite. (Specifically, we used a greedy test-suite reduction [48].) Using the minimal test suite, we minimized the mutant set to obtain the minimal mutant set for each project. The size of the automatically generated test pool and the size of the minimal mutant sets for each project are also in Table 4. Note that by definition, the number of mutants in the minimal mutant set is the same as the number of tests in the minimal test suite.

3.5.5 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.) We then create a reduced test case that preserves these $N$ selected mutants. In addition to randomly selecting $N$ mutants for each test case, we also select to preserve the mutant(s) from the minimal mutant set that a test case kills. We choose to reduce each test case in the minimal test suite and to reduce it to preserve the mutant that it contributes to the minimal mutant set (as described in Section 3.5.4). Because minimal mutants are defined with respect to the minimal test suite, we only reduce test cases in the computed minimal test suite. Moreover, a property of minimal mutants is that each test case in the minimal test suite uniquely kills exactly 1 mutant in the minimal mutant set, so the value $N = 1$ always here.

Performing test reduction can take a long time for some test cases. We limit the 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. We do not use test cases whose reduction times out in the experiments.

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 a baseline for comparison; we do not necessarily recommend using random test-case reduction as a means to create smaller test cases because our analysis shows that such randomly reduced test cases have lower quality than test cases reduced by our $C\%-coverage$ or $N$-mutant reductions.

For each reduced test case, we finally measure the three metrics of SRR, CPR, and MPR. For SRR, we measure the number of parts in the reduced test case and compare it with the number of parts in the original test case, as per Definition 3. For CPR, we run both the original test case and the reduced test case using gcov to obtain how many statements are covered, and then use Definition 4. Similarly, for MPR, we run both the original test case and the reduced test case using the mutation-testing tool [2], and then use Definition 5; to speed up experiments, we only run with the mutants killed by the original test case to find which of those mutants are killed by the reduced test case. In other words, we do not measure whether the reduced test case kills some mutants that the original test case did not kill.

3.6 Research Questions

In our evaluation, we address the following four research questions. We report the result of first research question in this report.

- RQ1: How much can test cases be reduced (SRR) in non-adequate test-case reduction?
- RQ2: How much code coverage and mutants killed are preserved (CPR and MPR) in non-adequate test-case reduction?
- RQ3: What are the trade-offs between SRR, CPR, and MPR?
- RQ4: How do CPR and MPR in non-adequate test-case reduction compare to CPR and MPR in random test-case reduction?

RQ1: SRR

Figures 3 and 4 summarize the results for SRR on the reduced test cases obtained using $C\%-coverage$ and $N$-mutant, respectively. For each subject and each level of $C$ and $N$, the boxplots show the distribution of SRR.

4. REMAINING WORK AND TIMELINE

Figure 5 illustrates high-level goals of this research. The first goal is to use the information in generated tests and devise a technique for focused random testing, which is realized by directed swarm testing. Goal 2 is to try to reduce size of test cases while preserving (most of) the properties of the original test cases, which will be achieved by non-adequate reduction. In Goal 3, we intend to explore the benefits of using the reduced test cases reduced by non-adequate reduction as seeds to dynamic symbolic execution.

4.1 Impact of (nonadequate)reduced test cases on exploration of symbolic execution
We want to study the impact of using test cases reduced by non-adequate test reduction as seeds to dynamic symbolic execution. This study will be similar to our ISSTA’14 [50] study, but we will examine the use non-adequate test reduction instead of 100%-Coverage for reducing test cases.

### 4.2 Implementing Random Focused Testing for Java in Randoop

Randoop is a popular random testing tool for Java programs. A natural extension of focused testing would be to adapt it for Java programs and compare it with various optimizations that Randoop has provided. We intend to extend Randoop with features for swarm testing, with particular emphasis on directed swarm testing.

### 4.3 Project Timing

Table 5 illustrates the venues that we will target to publish the results of this research and our projected completion dates. We need an additional three month to clean up the code, document the artifacts, and write the dissertation. The expected finish date is Spring 2017.

### 5. DISCUSSION

By using the tests to generate new test cases, or to direct test generation, we attempt to prove that generated tests are software artifacts worth processing. We expect this dissertation to improve our understanding of the extent to which information in generated tests can be used in software engineering, particularly in the area of software testing.

### 6. REFERENCES


Appendix–Research Portfolio

Journal Articles


Conference and Workshop Papers

2. Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the Limits of Mutation Reduction Strategies. ACM/IEEE International Conference on Software Engineering (ICSE’16), Austin, Texas, May 2016 (acceptance rate 19%).


4. Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How Hard Does Mutation Analysis Have to Be, Anyway? IEEE International Symposium on Software Reliability Engineering (ISSRE’15), Gaithersburg, Maryland, November 2015 (acceptance rate 32%).


6. Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using Test Case Reduction and Prioritization to Improve Symbolic Execution. ACM International Symposium on Software Testing and Analysis (ISSTA’14), pages 60–70, San Jose, California, July 2014 (acceptance rate 28%).


