Generating Focused Random Tests using Directed Swarm Testing

Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, Arpit Christi
School of Electrical Engineering and Computer Science
Oregon State University, United States
{alipour,alex,gopinath, christie}@eecs.oregonstate.edu

ABSTRACT

Random testing can be a powerful and scalable method for finding faults in software. However, sophisticated random testers usually test a whole program, not individual components. Writing random testers for individual components of complex programs may require unreasonable effort. In this paper we present a novel method, *directed swarm testing*, that uses statistics and a variation of random testing to produce random tests that focus on only part of a program, increasing the frequency with which tests cover the targeted code. We demonstrate the effectiveness of this technique using real-world programs and test systems (the YAFFS2 file system, GCC, and Mozilla's SpiderMonkey JavaScript engine), and discuss various strategies for directed swarm testing. The best strategies can improve coverage frequency for targeted code by a factor ranging from 1.1-4.5x on average, and from nearly 3x to nearly 9x in the best case. For YAFFS2, directed swarm testing never decreased coverage, and for GCC and SpiderMonkey coverage increased for over 99% and 73% of targets, respectively, using the best strategies. Directed swarm testing improves detection rates for real SpiderMonkey faults, when the code in the introducing commit is targeted. This lightweight technique is applicable to existing industrial-strength random testers.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

random testing, regression testing, swarm testing

1. INTRODUCTION

Random testing [19] (sometimes called fuzzing) is now widely recognized as an effective approach for testing software systems, including compilers [31, 32, 37], standard libraries [29], static analysis systems [7], and file systems [16]. 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 [4]. 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 [38]. It is seldom practical to implement different random testers for all the potential focuses that might be needed, and many powerful random testers [16, 31, 37] 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 [13], 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 [23]). 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 [11, 36] and search-based techniques [20, 27]) 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,

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany

http://dx.doi.org/10.1145/2931037.2931056

[1]SAGE [10] for symbolic execution, and the work of Kifetew et al. [22] 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`.

including C compilers [25, 37], JavaScript engines [21, 31, 32], and Google's Go language [34]. 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 [18] (a variation of random testing) and recording statistical data on past testing results [17] 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 [37] 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 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 3).
- Strategies for targeting both individual source code targets and multiple source code targets at once (Section 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 5 and 6).
- Empirical results of effectiveness of these strategies on finding *real faults* in a large software system (Sections 5 and 6).

2. PRELIMINARY CONCEPTS

Swarm testing [18] 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 [1, 12]). 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 [25] and has sometimes nearly *doubled* fault detection and/or coverage for mature random testers [17]. Swarm testing has also been applied to the CCG C compiler testing tool [28] and the GoSmith [34] 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 [9].

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 considers 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 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

```
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 1: Features for Random Test Cases

configuration, a tester can usually generate an unbounded number of different tests containing (at most) those features.

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 [18].

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 [17], 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 [35] 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 [17].

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 [13], 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 compute triggers and successors for a program. Running

[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.

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 [17]. 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 [13, 14, 33].

4. CONFIGURATION STRATEGIES

Figure 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 2.1) and in more detail in the empirical work of Groce et al., [17], 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.

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 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.

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

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

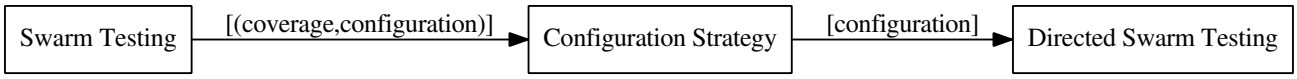

Figure 2: Workflow of directed swarm testing.

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 [30]. We implemented an SMT-based exact solver for merging targets using Z3 [8], 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 [17], 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.

5. 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 [16], and applied in previous work on combining random testing and symbolic execution [38]. For GCC, we used the Csmith [37] 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` [31], 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 [37] 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

Table 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

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 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 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.

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

[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.

Table 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

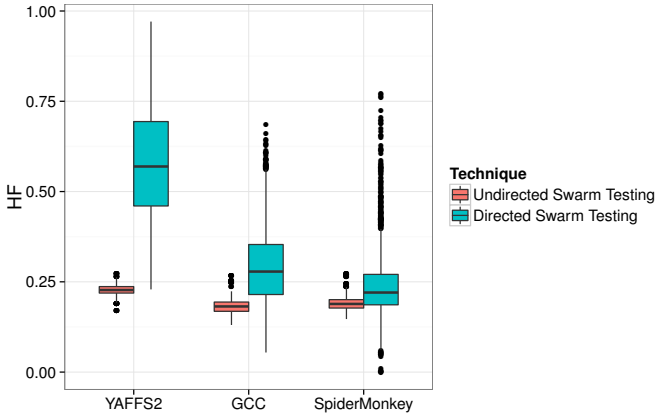

Figure 3: Hitting fraction in undirected swarm testing (HF_u) versus directed swarm testing (HF_d) over all strategies.

$\frac{HF_d}{HF_u} > 1.0$ offer improvement over undirected test suites. This is the measure a developer wants to increase via targeting.

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 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 3 provides much more detailed information about the performance of directed testing for single-target directed testing[5]. It

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

Table 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

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.

6.1 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. Figure 4 visualizes these results. 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.

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.

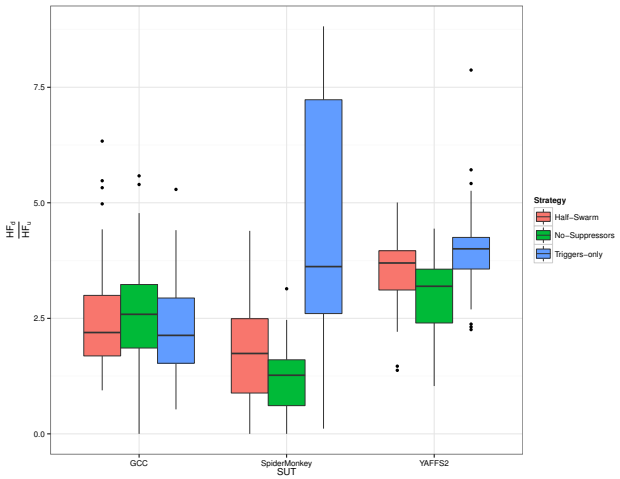

Figure 4: Single-target strategies compared.

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.

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 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.

[6]Full statistical test results in tech report.

Table 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 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).

Figure 6 shows the performance of different merge strategies across test subjects, and Figure 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.

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 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.

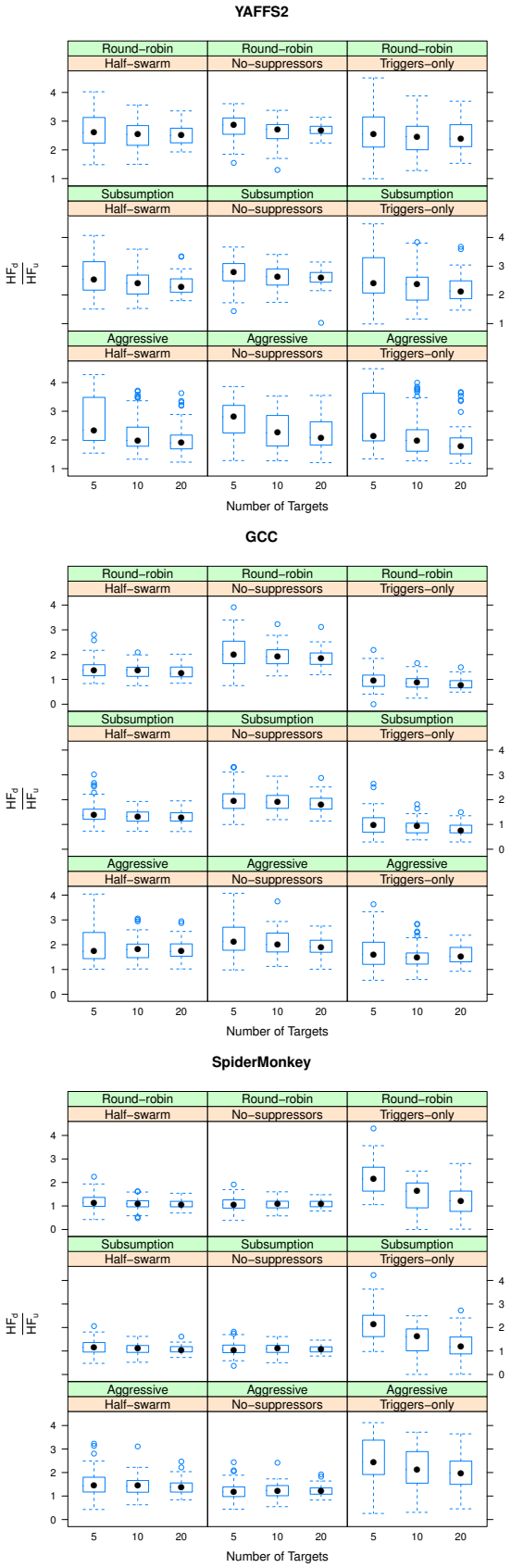

Figure 5: Multi-target strategies compared.

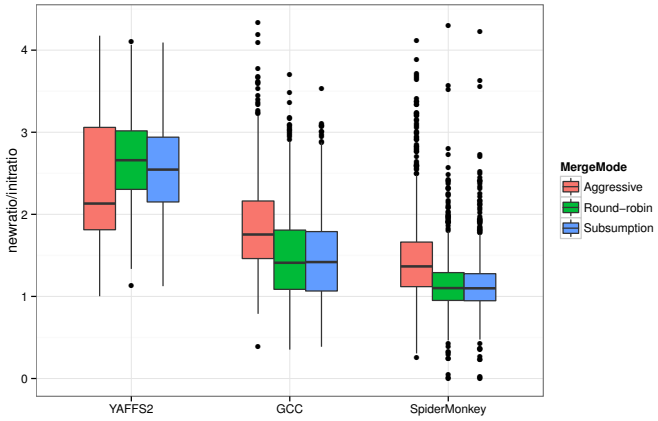

Figure 6: Merge strategies over all multi-target strategies.

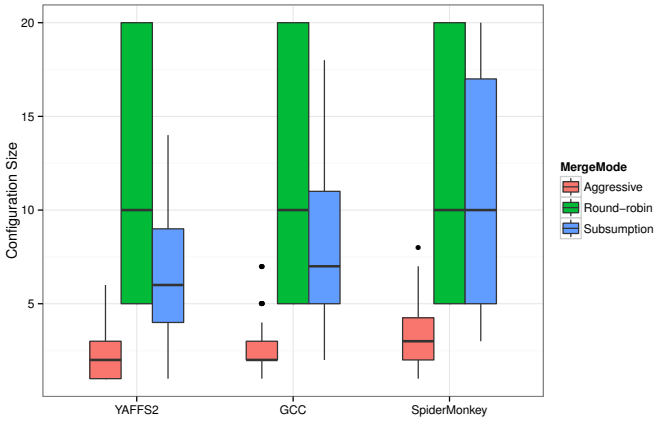

Figure 7: Number of targets after merging, by merge strategy.

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.

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 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) [16]. 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 8 illustrates the results. We used a paired t-test between HF_d and HF_r. Table 5 summarizes the results, showing average hitting fractions, confidence intervals on the effect size, and p-values.

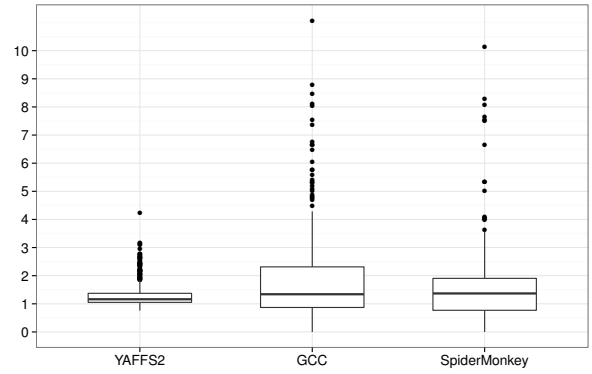

Figure 8: $\frac{HF_d}{HF_r}$, random vs. directed.

Table 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.

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 [26]) aim to hit targets only once.

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 (usually 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 [18]). 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 [18]. 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 [12]. 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).

9. RELATED WORK

The most closely related work is our previous work introducing swarm testing [18] and the notions of triggers and suppressors [17]. 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 [20,27] and (dynamic) symbolic execution [11,36] are the most notable ones. Symbolic execution [24] 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 [11]. 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 [20, 27].

There are many previous efforts to improve random testing. Randoop [29] generates tests for object-oriented programs by calling random APIs, but uses feedback to guide test sequence creation. Nighthawk [2] 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 [5, 6] aims to improve random testing by using a distance measure to select more uniformly distributed tests, though its effectiveness in practice has been criticized [3]. ABP-based testing uses reinforcement learning to guide test generation [15].

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 [38]. 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.

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.

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.

12. REFERENCES

[1] ALIPOUR, M. A., AND GROCE, A. Bounded model checking and feature omission diversity. In *International Workshop on Constraints in Formal Verification* (2011).

[2] ANDREWS, J. H., LI, F. C. H., AND MENZIES, T. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 144–153.

[3] ARCURI, A., AND BRIAND, L. Adaptive random testing: An illusion of effectiveness. In *International Symposium on Software Testing and Analysis* (2011), pp. 265–275.

[4] ARCURI, A., IQBAL, M. Z. Z., AND BRIAND, L. C. Formal analysis of the effectiveness and predictability of random testing. In *International Symposium on Software Testing and Analysis* (2010), pp. 219–230.

[5] CHEN, T. Y., KUO, F.-C., MERKEL, R. G., AND TSE, T. H. Adaptive random testing: The art of test case diversity. *J. Syst. Softw. 83*, 1 (Jan. 2010), 60–66.

[6] CHEN, T. Y., LEUNG, H., AND MAK, I. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2005, pp. 320–329.

[7] CUOQ, P., MONATE, B., PACALET, A., PREVOSTO, V., REGEHR, J., YAKOBOWSKI, B., AND YANG, X. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium* (2012), pp. 120–125.

[8] DE MOURA, L. M., AND BJØRNER, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.

[9] DEWEY, K., ROESCH, J., AND HARDEKOPF, B. Fuzzing the rust typechecker using CLP. In *Automated Software Engineering* (2015), pp. 482–493.

[10] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 206–215.

[11] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

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

[13] GROCE, A., ALIPOUR, M. A., ZHANG, C., CHEN, Y., AND REGEHR, J. Cause reduction for quick testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on* (2014), IEEE, pp. 243–252.

[14] GROCE, A., ALIPOUR, M. A., ZHANG, C., CHEN, Y., AND REGEHR, J. Cause reduction: delta debugging, even without bugs. *STVR 26*, 1 (2015), 40–68.

[15] GROCE, A., FERN, A., PINTO, J., BAUER, T., ALIPOUR, A., ERWIG, M., AND LOPEZ, C. Lightweight automated testing with adaptation-based programming. In *IEEE International Symposium on Software Reliability Engineering* (2012), pp. 161–170.

[16] GROCE, A., HOLZMANN, G., AND JOSHI, R. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering* (2007), pp. 621–631.

[17] GROCE, A., ZHANG, C., ALIPOUR, M., EIDE, E., CHEN, Y., AND REGEHR, J. Help, help, I'm being suppressed; The significance of suppressors in software testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (Nov 2013), pp. 390–399.

[18] GROCE, A., ZHANG, C., EIDE, E., CHEN, Y., AND REGEHR, J. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 78–88.

[19] HAMLET, R. Random testing. In *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[20] HARMAN, M., MANSOURI, S. A., AND ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv. 45*, 1 (Dec. 2012), 11:1–11:61.

[21] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 38–38.

[22] KIFETEW, F. M., TIELLA, R., AND TONELLA, P. Combining stochastic grammars and genetic programming for coverage testing at the system level. In *Search-Based Software Engineering* (2014), pp. 138–152.

[23] KIM, S., WHITEHEAD, E., AND ZHANG, Y. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on 34*, 2 (March 2008), 181–196.

[24] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[25] LE, V., AFSHARI, M., AND SU, Z. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 216–226.

[26] MARINESCU, P. D., AND CADAR, C. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, pp. 235–245.

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

[28] NAGAI, E., HASHIMOTO, A., AND ISHURA, N. 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* (2013), pp. 88–93.

[29] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 75–84.

[30] ROBERTSON, E. L., AND WYSS, C. M. Optimal tuple merge in NP-complete. Tech. Rep. TR599, Indiana University Bloomington, July 2004.

[31] RUDERMAN, J. Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[32] RUDERMAN, J. Releasing jsfunfuzz and DOMFuzz. https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2015.

[33] SHI, A., GYORI, A., GLIGORIC, M., ZAYTSEV, A., AND MARINOV, D. Balancing trade-offs in test-suite reduction. In *FSE* (2014), pp. 246–256.

[34] VYUKOV, D. gosmith: Random Go program generator. https://code.google.com/p/gosmith/.

[35] WILSON, E. B. Probable inference, the law of succession, and statistical inference. *J. of the American Statistical Assoc. 22* (1927), 209–212.

[36] XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on* (2009), IEEE, pp. 359–368.

[37] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 283–294.

[38] ZHANG, C., GROCE, A., AND ALIPOUR, M. A. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis* (2014), pp. 160–170.