

Swarm Testing

Alex Groce,* Chaoqiang Zhang,* Eric Eide,† Yang Chen,† and John Regehr†

*School of Electrical Engineering and Computer Science

Oregon State University, Corvallis, OR

Email: agroce@gmail.com, zhangch@onid.orst.edu

†School of Computing

University of Utah

Email: {[eeide](mailto:eeide@cs.utah.edu), [chenyang](mailto:chenyang@cs.utah.edu), [regehr](mailto:regehr@cs.utah.edu)}@cs.utah.edu

Abstract—Swarm testing is a novel and inexpensive way to improve the diversity of test cases generated during random testing. Increased diversity leads to improved coverage and fault detection. In swarm testing, the usual practice of potentially including all features in every test case is abandoned. Rather, a large “swarm” of randomly generated configurations, each of which omits some test features, is used, with configurations receiving equal resources. Omitting features can enable better exploration of the state space of a system. First, some features actively prevent the system from reaching interesting parts of its state space; e.g., “pop” calls may prevent a stack data structure from executing a bug in its overflow detection logic. Second, even when there is no active suppression of behaviors, test features compete for space in each test, limiting the depth to which logic driven by features can be explored. Experimental results show that swarm testing increases coverage and can improve fault detection dramatically; for example, in a week of testing it found 42% more distinct ways to crash a collection of C compilers than did the heavily hand-tuned default configuration of a random tester.

Keywords-random testing; configuration diversity

I. INTRODUCTION

This paper focuses on answering a single question: In random testing, can a diverse set of *testing configurations* perform better than a single, possibly “optimal” configuration? A configuration would be represented, for example, by the frequency with which each feature would be included in test cases. Conventional wisdom in random testing [1] has assumed a policy of (1) finding a “good” configuration and (2) running as many tests as possible under that configuration. Considerable research effort has been devoted to the question of how to tune a “good configuration,” e.g., how to use genetic algorithms to optimize the frequency of various method calls in random tests [2], or how to choose a length for random test runs [3]. As a rule, the notion that some configurations are “good” and that finding a good (if not optimal, given the size of the search space) configuration is one key to the art of successful random testing has not been challenged. Furthermore, in the interests of maximizing coverage and fault detection, it has been assumed that a good random test configuration includes as many API calls or other input domain features as possible, and this has been the guiding principle in large-scale successful efforts to test

C compilers [4], file systems [5], and utility libraries [6]. The rare exceptions to this rule have been cases where a feature makes tests too difficult to evaluate or slow to execute, or when static analysis or hand inspection can demonstrate that an API call neither depends on nor alters system state [5]. For example, a file system test may not include calls to check free space on a volume because the space usage model for a reference system cannot predict the correct return value, or predicate assertions may make compiling random C programs significantly slower with certain compilers.

In general, if a call or feature is omitted from some tests, it is usually omitted from all tests. This approach seems to make intuitive sense: omitting features, unless it is necessary, means *giving up on detecting some faults and covering parts of the state space*. However, this objection to feature omission only holds so long as testing is performed using a single, “optimal” configuration. Swarm testing, in contrast, uses a diverse “swarm” set of test configurations, each of which *deliberately omits certain API calls or input features*. The result is that, given a fixed testing budget, swarm testing tends to test a more diverse set of inputs than would be tested under a so-called “optimal” configuration (perhaps better referred to as a *default* configuration) in which every feature is available for use by every test.

One can visualize the impact of swarm testing by imagining a “test space” defined by the features utilized by tests. As a simple example, consider the job of testing an implementation of a stack ADT that provides two operations, push and pop. One can define a test space for the stack using these features as axes: an individual test is characterized by the number of times it invokes each operation. A given method for randomly generating test cases results in a probability distribution over the test space, with the value at each point (x, y) giving the probability that a given test will contain exactly x pushes and y pops (in any order). To make this example more interesting, imagine that the stack under test has a capacity bug, and that it will crash whenever the stack is required to hold more than 32 items.

Figure 1(a) illustrates the situation for testing the stack with a test generator that chooses pushes and pops with equal probability. The generator randomly chooses an input

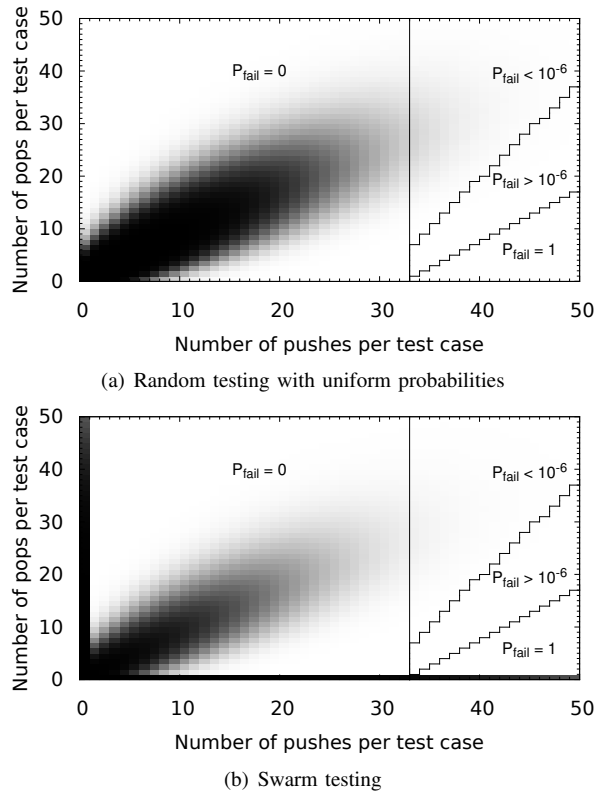


Figure 1. Swarm testing changes the distribution of test cases for a stack. If push and pop operations are selected with equal probability, about 1 in 370,000 test cases will trigger a bug in a 32-element stack’s overflow-detection logic. Swarm testing triggers this bug in about 1 of every 16 cases.

length and then decides if each operation is a push or a pop. The graph shows the distribution of tests produced by this generator over the test space. The graph also shows contour lines for significant regions of the test space. Where $P_{fail} = 1$, a test chosen randomly from that region is certain to trigger the stack’s capacity bug; where $P_{fail} = 0$, no tests can trigger the bug. As Figure 1(a) shows, this generator only rarely produces test cases that can trigger the bug.

Now consider a test generator based on swarm testing. This generator first chooses a non-empty subset of the stack API and then generates a test case using that subset. Thus, one-third of the test cases contain both pushes and pops, one-third just pushes, and one-third just pops. Figure 1(b) shows the distribution of test cases output by this generator. As is evident from the graph, this generator often produces test cases that will trigger the stack’s capacity bug.

Although simple, this example illustrates the dynamics that make swarm testing work. The low dimensionality of the stack example is contrived, of course, and we certainly believe that programmers should make explicit efforts to test boundary conditions. As evidenced by the results presented in this paper, however, swarm testing generalizes to real situations in which there may be dozens of features that can be independently turned on or off. It also generalizes to

testing real software in which faults are very well hidden.

Every test generated by any swarm configuration can, in principle, be generated by a test configuration with all features enabled. However—as the stack example illustrates—the probability of covering parts of the state space and detecting certain faults can be demonstrably higher when a diverse set of configurations is tested.

Swarm testing has several important advantages. First, it is low cost: in our experience, existing random test case generators already support or can be easily adapted to support feature omission. Second, swarm reduces the amount of human effort that must be devoted to tuning the random tester. In our experience, tuning is a significant ongoing burden. Finally—and most importantly—swarm testing makes significantly better use of a fixed CPU time budget than does random testing using a single general-purpose configuration, in terms of both coverage and fault detection. For example, we performed an experiment where two machines, differing only in that one used swarm testing and one did not, used test cases generated by Csmith [4] to attempt to crash a collection of 17 production-quality C compilers for x86-64. During one week of testing, the swarm machine found 104 distinct ways to crash a compiler in the test suite whereas the other machine—running the default Csmith configuration which enables all features—found only 73. An improvement of more than 40% in terms of number of bugs found, when compared to a random tester that has been intensively tuned for several years, is surprising and significant.

Even more surprising were some of the details. We found, for example, a compiler bug that could only be triggered by programs containing pointers, but which was almost never triggered by inputs that contained arrays. This is odd because pointer dereferences and array accesses are very nearly the same thing in C.¹ Moreover, we found another bug in the same compiler that was only triggered by programs containing arrays, but which was almost never triggered by inputs containing pointers. Fundamentally, it appears that omitting features while generating random test cases can lead to improved test effectiveness.

Our contributions are as follows. First, we characterize *swarm testing*, a pragmatic variant of random software testing that increases the diversity of generated test cases with little implementation effort. The swarm approach to diversity differs from previous methods in that it focuses solely on *feature omission diversity*: variance in which possible input features are *not* present in test cases. Second, we show that—in two case studies—swarm testing offers improved coverage and bug-finding power. Third, we offer some explanations as to *why* swarm testing works.

II. SWARM TESTING

Assume that a *test configuration* C is a valuation for a set of Boolean feature variables, $f_1 \dots f_n$. Features may be

¹In C/C++, $a[i]$ is syntactic sugar for $*(a+i)$.

API calls (e.g., push and pop) or attributes of an input that a test generation algorithm will have direct control over (e.g., whether a C program contains array accesses, or whether a media player file has a corrupt header). In the remainder of the paper, we will use *feature* to represent API calls and other attributes of inputs. C is used as the input to a random testing function [1], [7] $gen(C, s, l)$, which given configuration C , seed s , and length l , generates a test case containing l calls chosen from the enabled features in C , for the Software Under Test (SUT). We may ignore the details of how allowed features are distributed and how feature attributes are selected. The values for $f_1 \dots f_n$ determine which features are allowed to appear in the test case. For example, if we are testing a simple file system, the set of features may be: $\{ read, write, open, close, unlink, sync, mkdir, rmdir, unmount, mount \}$. A typical default configuration would then be $\{ read, write, open, close, unlink, sync, mkdir, rmdir \}$, which avoids calling mount and unmount, in order to avoid wasting test time on operations while the file system is unmounted. Assume that a test engineer has 24 hours to test a file system, and two CPUs available. The conventional strategy would be to choose a “good” test case length, divide the set of random-number-generator seeds into two sets (e.g., odd and even values) and simply generate, execute, and evaluate as many tests as possible on each CPU.

In contrast, a swarm approach to testing the same system, under the same assumptions, would use a “swarm”—a fixed set of randomly chosen unique configurations, all with *mount* and *unmount* set to false and other features having probability 50% of being selected—and then divide the total time budget on each CPU such that each configuration in the swarm received equal testing time. In this case, where there are nine features and thus 2^9 (512) possible configurations, the swarm set might consist of 64 configurations, each of which would receive a test budget of 45 minutes (two days of CPU time divided by 64 configurations). Some configurations (those omitting features that slow down the SUT) might execute more tests during 45 minutes of testing than the default configuration; others might execute fewer tests, due to a high concentration of expensive features. On average, the total number of tests executed will be similar to the standard approach, though perhaps with a greater variance. The distribution of calls made, over all tests, will also be similar to that found using the default configuration: each call will be absent in roughly half of all configurations, but will be called more frequently in other configurations. Why, then, might results from the swarm approach differ from those under the default approach?

A. Disadvantages of Configuration Diversity

An immediate objection to swarm testing is that it may make certain faults impossible to detect. Consider a file system bug that can only be exposed by a combination of

calls to *read*, *write*, *open*, *mkdir*, *rmdir*, *unlink*, and *sync*. Because there is only a 1/128 chance that a given configuration will enable all of these calls, it is likely that a set of 64 configurations will not contain any that could produce a test case exposing this fault. Furthermore, in the case of file systems, an intelligent engineer will note that calling *read* and *write* in test cases that do not contain *open* calls is likely to be of relatively little value. At first examination, it seems likely that the swarm approach to testing will expose fewer faults and produce worse coverage of the code and state space than the default approach to testing. Furthermore, recalling that any test possible with any swarm configuration is also possible under the default configuration, but that some tests produced by the default configuration may be impossible to produce, for many swarm sets, it may be hard to imagine how swarm can compensate for these problems.

B. Advantages of Configuration Diversity

The key insight is that the *possibility* of a test being produced is not the same as the *probability* that it will be produced. In particular, consider a fault that relies on making 64 calls to *open*, without any calls to *close*, at which point the file descriptor table overflows and the file system crashes. If the test length in both the default and swarm settings is fixed at 512 operations, we know that the default configuration is highly unlikely to expose the fault (the rate is much less than 1 in 100,000 tests). With swarm, on the other hand, many configurations (16 on average) will produce tests containing calls to *open* but no calls to *close*. Furthermore, some of these configurations will also disable other calls, increasing the proportion of *open* calls made. For configurations enabling *open*, and disabling *close* and at least one other feature, the probability of detecting the fault improves from close to 0% to over 80%. If 48 hours of testing produces approximately 100,000 tests, it is almost certain that the default configuration will fail to detect this fault, and at the same time almost certain that any set of swarm configurations will detect the fault. The same argument holds even if we improve the chances of the default configuration by assuming that *close* calls do not decrement the file descriptor count: swarm is more likely to produce any failure that requires a large number of calls to any one function in a single test.

While such resource-exhaustion faults may seem to be a rare special case, the concept can be generalized. If we assume that some aspects of system states are affected more by some features than others, then by shifting the distribution of calls within each test case (though not over all test cases), swarm testing results in a much higher probability of exploring “deep” values of some state variables. Only if all features equally affect all state variables are swarm and the default configuration approach equally likely to explore deep states. Given that real systems exhibit a strong degree of

modularity and that API calls and input features are typically *designed* to have predictable, localized effects on system state or behavior, this seems extremely unlikely. Moreover, the problem of `close` masking the problem with `open` is also generalizable: e.g., in the file system setting, it seems likely that many faults related to buffering will be masked by calls to `sync`. In a compiler, many potentially faulty optimizations will never be applied to code that contains pointer accesses, because of failed safety checks based on aliasing. In other words, including a feature in a test does not always improve the ability of the test to cover code and state and expose faults: some features can suppress the exhibition of some behaviors. We therefore need tests that exhibit a high degree of feature omission diversity, since we do not know which features will suppress behaviors, and features are almost certain both to suppress some behaviors and be required for other behaviors.

C. An Empirical Question

In general, all that can be said is that what we call the default (maximally inclusive) configuration may be optimal for some hypothetical set of coverage targets and faults, while a set of swarm configurations will contain configurations that are optimal for more (again, hypothetical) coverage targets and faults, but will perform less testing optimized for each target than the conventional approach will perform on its targets. Our hypothesis is that for many real-world programs, the conventional, single-configuration approach to testing will expose the same faults and cover the same behaviors many times, while swarm testing may expose more faults and cover more targets, but might well produce fewer failing tests for each fault and execute fewer tests that cover each branch/statement/path/etc. However, given that the precise distribution of faults and coverage targets in the state space of any realistic system is extremely complex and is not amenable to *a priori* analysis, only experimental investigation of how the two testing approaches compare on real systems can give us valid insight into what strategies might be best for large-scale (random) testing of critical systems. The remainder of this paper shows how conventional default configuration testing and swarm testing compare for coverage and fault detection on a widely used open-source flash file system and a large number of widely used C compilers.

D. Evaluating Swarm Testing

The thesis of this paper is that randomly turning off features during test case generation can lead to more effective random testing. Thus, one of our evaluation criteria is that swarm should find more defects and lead to improved code coverage, when compared to the default configuration of a random tester, with other factors being equal.

In the context of any individual bug, it is possible to evaluate swarm in a more detailed fashion by analyzing the

features found *and not found* in test cases that trigger the bug. We say that a test case feature is *significant* with respect to some bug if its presence or absence affects the likelihood that the bug will be found. For example, push operations are obviously a significant feature with respect to the example in Section I because a call to push causes the bug to manifest. But this is banal: it has long been known that effective random testing requires “featureful” test inputs. The power of swarm is illustrated only when the *absence* of one or more features is statistically significant in triggering a bug. Section III shows that significant negative (suppressing) features for bugs are commonplace, providing—we believe—strong support for our claim that swarm testing is useful.

III. CASE STUDIES

We evaluated swarm testing using a file system and a collection of 17 production-quality C compilers. The file system was small enough (15 KLOC) to be subjected to mutation testing. The compilers, on the other hand, were not conveniently sized for mutation testing, but provided something better: a large set of real faults that caused crashes or other abnormal exits.

A. Case Study: YAFFS Flash File System

1) *Background and Methodology:* YAFFS2 [8] is a popular open-source NAND flash file system for embedded use; it is the default image format for the Android operating system. Our test generator for YAFFS2 produces random tests of any desired length and executes the tests using the RAM flash emulation mode. By default, tests can include any of 23 core API calls, appearing with generally equal probability. Feedback [5], [6] is used, for example, to ensure that calls such as `close` and `readdir` occur only in states where valid file descriptors or `dirent` objects are available. The tester also allows us to disable any number of calls in a test. The 23 YAFFS2 calls tested constituted the feature set in our YAFFS2 experiments. Swarm configurations were produced by disabling API calls with 50% probability. We ran one experiment with 100 configurations and another with 500 configurations. Both swarms are large enough compared to the number of tests run to make unusual effectiveness or poor performance due to a small set of especially good or bad configurations highly unlikely. Both experiments compared 72 hours of swarm testing to 72 hours of testing with the default configuration (with all 23 API calls enabled), evaluating test effectiveness based on block, branch, du-path, prime path, path, and mutation coverage [9]. For prime and du-paths, we limit lengths to a maximum of 10. Path coverage was measured at the function level (that is, a path is the path taken from entry to exit of a single function). Both experiments used 532 mutants to evaluate mutation coverage. These mutants were not guaranteed to be killable by the API calls and emulation mode used (we expect that about half of these mutants lie

Table I
YAFFS2 COVERAGE RESULTS

	swarm 100 mutants online			swarm 500 mutants offline		
	Def.	Swarm	Both	Def.	Swarm	Both
#	1,747	1,593	3,340	5,665	5,888	11,553
bl	1,161	1,168	1,173	1,173	1,172	1,178
br	1,247	1,253	1,261	1,261	1,259	1,268
du	2,487	2,507	2,525	2,525	2,538	2,552
pr	2,834	2,872	2,964	2,907	2,967	3,018
pa	14,153	25,484	35,478	35,432	64,845	91,280
mu	94	97	97	95	97	97

= number of tests performed; bl = blocks covered; br = branches covered; du = du-paths covered; pr = prime paths covered; pa = paths covered; mu = mutants killed

in unreachable code, and some portion of the remainder are semantically equivalent to the original YAFFS2). The only difference between experiments, other than the number of swarm configurations, was that in the first experiment mutation coverage was computed online, and was included in the test budget for each approach. The second experiment did not count mutation coverage computations as part of the test budgets, but executed all mutation tests offline, in order to show how results changed with increased test throughput.

2) *Results—Coverage and Mutants Killed:* Table I shows how swarm configuration affects testing of YAFFS2. For each experiment, the first column of results shows how the default configuration performed, the second column shows the coverage for swarm testing, and the last column shows the coverage for combining the two test suites. Each individual test input contained 200 file system operations. Therefore, the swarm test suite in the second experiment executed a total of 1.17 million operations, and testing all mutants required an additional 626 million YAFFS2 calls.

The first experiment (columns 2–4 of Table I) shows that, despite executing 154 fewer tests, swarm testing improved on the default configuration in *all* coverage measures—the difference is particularly remarkable for path coverage, where swarm testing explored over 10,000 more paths. The combined test suite results show that default and swarm testing largely overlapped in coverage, but that default configuration did explore some blocks, branches, and paths that swarm testing did not. Surprisingly, swarm testing was *strictly superior* in terms of mutation kills: swarm killed three mutants that the default configuration did not, and killed every mutant killed by the default configuration. On average, the default configuration killed each mutant 1,173 times. The swarm configuration only killed each mutant (counting only those killed by both test suites, to avoid any effect from swarm also killing particularly hard-to-kill mutants) an average of 725 times. An improvement of three mutants out of 94 may seem small, but a better measure of fault detection capabilities may be kill rates for nontrivially detectable mutants. It seems reasonable to consider any

mutant killed by more than 10% of random tests (each with only 200 operations!) to be uninteresting. Even quite desultory random testing will catch such faults. Of the 97 mutants killed by the default configuration, only 14 were killed by < 10% of tests, making swarm’s 17 nontrivial kills an improvement of over 20%.

In the second experiment (columns 5–7 of Table I), test throughput was about 3x greater due to offline computation of mutation coverage. The default configuration covered slightly more blocks (1) and branches (2) than the swarm tests. However, of the six blocks and nine branches covered only by the default configuration, all but four (two of each) were low-probability behaviors of the `rename` operation. In file system development and testing at NASA’s Jet Propulsion Laboratory, `rename` was by far the most complex and faulty operation, and we expect that this is true for many file systems [5]. This result suggests a vulnerability (or intelligent use requirement) for swarm testing: if a single feature is expected to account for a large portion of the behavioral complexity and potential faults in a system, it *may* be best to set the probability of that feature to more than 50%. Nonetheless, swarm managed to produce better coverage for all other metrics, executing almost 30,000 more paths than testing under the default configuration, and *still killed all of the mutants killed by the default configuration, and two additional mutants*. In fact, the additional 4,295 tests (with a completely different, and larger, swarm configuration set) did not add mutants to the set killed by swarm, suggesting good fault detection ability for swarm on YAFFS2, even with a small test budget. The improvement in nontrivial mutant kills was 13% (15 vs. 17 kills). The default configuration once more tended towards “overkill,” with an average of 3,756 killing tests per mutant. Swarm testing only killed each mutant an average of 2,669 times.

3) *Results—Significant Features:* Figure 2 shows a delta-debugged [10] version of one of the swarm tests that killed mutant #62. Default configuration testing failed, in both experiments, to kill this mutant, which turns a < operator in YAFFS2’s chunk-marking code into a > operator. The original line of code at line 2 of `yaffs_UseChunkCache` is:

```
if (dev->srLastUse < 0 || dev->srLastUse > 100000000)
```

The mutant is:

```
if (dev->srLastUse > 0 || dev->srLastUse > 100000000)
```

The minimized test requires no operations other than `yaffs_open`, `yaffs_write`, `yaffs_read` and `yaffs_freespace`. The five tests killing this mutant in the first experiment were all produced by two configurations, both of which disabled the `close`, `lseek`, `symlink`, `link`, `readdir`, and `truncate` calls. The universal omission of `close`, in particular, probably indicates that this API call actively interferes with triggering this fault: it is difficult to perform the necessary write operations to expose the bug if files can be closed at any point in the

```

fd0 = yaffs_open("/ram2k/umtpaybhue",
  O_APPEND|O_EXCL|O_APPEND|O_TRUNC|O_RDWR|
  O_CREAT, S_IREAD);
yaffs_write(fd0, rdbuf, 9243);
fd2 = yaffs_open("/ram2k/iri",
  O_WRONLY|O_RDONLY|O_CREAT, S_IWRITE);
fd3 = yaffs_open("/ram2k/iri",
  O_WRONLY|O_TRUNC|O_RDONLY|O_APPEND,
  S_IWRITE);
yaffs_write(fd3, rdbuf, 5884);
yaffs_write(fd3, rdbuf, 903);
fd6 = yaffs_open("/ram2k/wz",
  O_WRONLY|O_WRONLY|O_CREAT, S_IWRITE);
yaffs_write(fd2, rdbuf, 3437);
yaffs_write(fd6, rdbuf, 8957);
yaffs_write(fd3, rdbuf, 2883);
yaffs_write(fd3, rdbuf, 4181);
yaffs_read(fd2, rdbuf, 8405);
fd12 = yaffs_open("/ram2k/gddlktknd",
  O_TRUNC|O_RDWR|O_WRONLY|O_APPEND|O_CREAT,
  S_IREAD);
yaffs_write(fd0, rdbuf, 3387);
yaffs_write(fd12, rdbuf, 2901);
yaffs_write(fd12, rdbuf, 9831);
yaffs_freespace("/ram2k/wz");

```

Figure 2. Operations in DD minimized test case for killing YAFFS2 mutant #62. The mutant returns an incorrect amount of remaining free space.

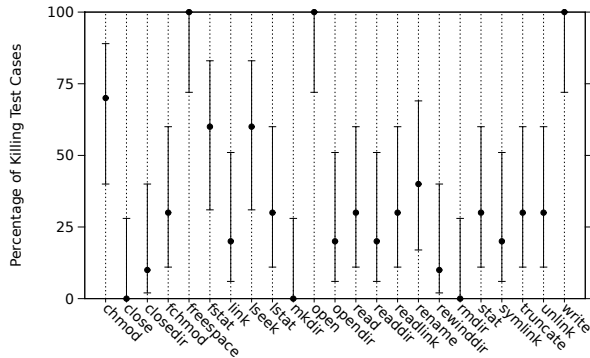


Figure 3. 95% confidence intervals for the percentage of test cases killing YAFFS2 mutant #62 containing each YAFFS2 feature. A value of one indicates the feature was present in all triggering test cases; a value of zero indicates it was always absent.

test. The other disabled options may all indicate passive interference: without omitting a large number of features it is difficult to explore the space of combinations of `open` and `write`, and observe the incorrect free space, in the 200 operations performed in each test. Figure 3 shows 95% confidence intervals for all YAFFS2 test features, computed over the configurations for the ten tests that killed this mutant in the second experiment. Calls to `freespace`, `open`, and `write` are clearly “triggers” (and almost certainly necessary) for this bug, while `close` is, as expected, an active suppressor. Calls to `mkdir` and `rmdir` are probably passive suppressors (we have not discovered any mechanism for active suppression, at least), as to a lesser degree are linking and directory read operations. As in model checking [11], turning off directory operations can give much better coverage of operations such as `write` and `read`.

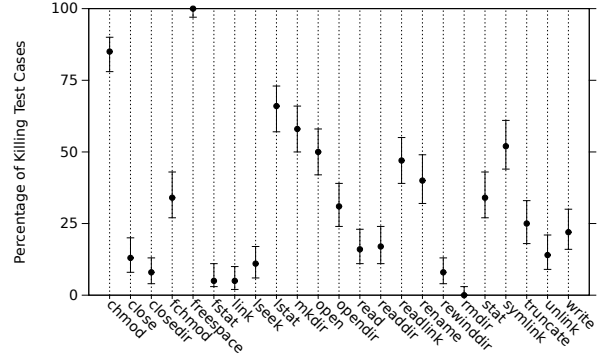


Figure 4. 95% confidence intervals for the percentage of test cases killing mutant #400 containing each YAFFS2 API call.

Figure 4 shows 95% confidence intervals for each feature in test cases killing another mutant that was only killed by swarm testing. This mutant negates a condition in the code for marking chunks dirty: `bi->pagesInUse == 0` becomes `bi->pagesInUse != 0`. Here, `freespace` is again required, but only `chmod` acts as a major trigger. This mutant affects a deeper level of block management, so either a file or directory can expose the fault: thus `mkdir` is a moderate trigger and `open` is possibly a marginal trigger but neither is required, as either call will set up conditions for exposure. A typical delta-debugged test case killing this mutant includes 15 calls to `chmod`, but `chmod` is not required to expose the fault—it is simply a very effective way to repeatedly force any directory entry to be rewritten to flash. Moreover, `rmdir` completely suppresses the fault—presumably by deleting directories before the `chmod`-induced problems can exhibit. It is possible to imagine hand-tuning of the YAFFS2 call mix helping to detect a fault like mutant #62. It is very hard to imagine a file system expert, one not already aware of the exact fault, tuning a tester to expose a bug like mutant #400.

Table II shows which YAFFS2 features contributed to mutant killing and which features suppressed mutant killing. Percentages represent the fraction of killed mutants for which the feature’s presence or absence was statistically significant. While mutants are not necessarily good representatives of real faults (we show how swarm performs for real faults below), the particular features that trigger and suppress the most bugs are quite surprising. For example, it is at first puzzling why `fchmod` is such a helpful feature. We believe this to be a result of the power of `fchmod` and `chmod` to “burn” through flash pages quickly by forcing rewrites of directory entries for either a file or a directory. The most likely explanation for `fchmod`’s value over `chmod` lies in `feedback`’s ability to always select a valid file descriptor, giving a rewrite of an open file; open files are more likely to be involved in faults, due to interactions with other features.

Table II also shows that *any* single configuration is likely

Table II
TOP TRIGGER AND SUPPRESSOR FEATURES IN SWARM TESTING OF
YAFFS2

Triggers		Suppressors	
fchmod	66%	rename	29%
lseek	65%	unlink	26%
read	62%	link	24%
write	61%	rewinddir	22%
open	58%	closedir	12%
close	57%	lstat	12%
fstat	56%	opendir	12%
symlink	41%	stat	11%
closedir	35%	mkdir	11%
opendir	35%	readdir	9%
truncate	34%	close	9%
rmdir	32%	symlink	8%
chmod	32%	truncate	8%
readdir	31%	rmdir	5%
mkdir	30%	chmod	5%
freespace	28%	fstat	4%
link	26%	lseek	4%
stat	25%	readlink	4%
readlink	18%	open	4%
unlink	16%	freespace	4%
lstat	14%	read	3%
rename	11%	write	3%
rewinddir	10%	fchmod	3%

Values in the table show the percentage of YAFFS2 mutants that were statistically likely to be triggered (killed) or suppressed by test cases containing the listed API calls.

to face challenges in YAFFS2 mutant killing. The three features that suppress the most faults are, respectively, also triggers for 11%, 16%, and 26% of mutants killed. The obvious active suppressors `close` and `closedir` are triggers for 57% and 35% of mutants killed, respectively.

B. Case Study: C Compilers

1) *Background:* Csmith [4] is a random C program generator; its use has resulted in about 400 bugs being reported to commercial and open-source compiler developers. Most of these reports have led to compiler defects being fixed. By default, Csmith errs on the side of expressiveness: it emits test cases containing all supported parts of the C language. To support swarm testing, we did not have to modify Csmith at all: it already had command line options for suppressing various features. These had previously been added by the Csmith developers (including ourselves) to support testing compilers for embedded platforms that only compile subsets of the C language, and for testing and debugging Csmith itself.

2) *Methodology:* We used Csmith to generate random C programs and fed these programs to 17 compilers targeting the x86-64 architecture; these compilers are listed in Table III. All of these tools are (or have been) in general use to compile production codes. All compilers were run under Linux. When possible, we used the pre-compiled binaries distributed by the compilers’ vendors.

Our testing focused on *distinct compiler crash errors*. This

Table III
DISTINCT CRASH BUGS FOUND DURING ONE WEEK OF TESTING

Compiler	Default	Swarm	Both
LLVM/Clang 2.6	10	12	14
LLVM/Clang 2.7	5	6	7
LLVM/Clang 2.8	1	1	1
LLVM/Clang 2.9	0	1	1
GCC 3.2.0	5	10	11
GCC 3.3.0	3	4	5
GCC 3.4.0	1	2	2
GCC 4.0.0	8	8	10
GCC 4.1.0	7	8	10
GCC 4.2.0	2	5	5
GCC 4.3.0	7	8	9
GCC 4.4.0	2	3	4
GCC 4.5.0	0	1	1
GCC 4.6.0	0	1	1
Open64 4.2.4	13	18	20
Sun CC 5.11	5	14	14
Intel CC 12.0.5	4	2	5
Total	73	104	120

metric considers two crashes of the same compiler to be distinct if and only if the compiler tells us that it crashed in two different ways. For example,

```
internal compiler error:
in vect_enhance_data_refs_alignment,
at tree-vect-data-refs.c:1550
```

and

```
internal compiler error:
in vect_create_epilog_for_reduction,
at tree-vect-loop.c:3725
```

are two distinct ways that GCC 4.6.0 can crash. We believe that this metric represents a conservative estimate of the number of true compiler bugs. Our experience—based on hundreds of bug reports to real compiler teams—is that it is almost always the case that distinct error messages correspond to distinct bugs. The converse is not true: many different bugs may hide behind a generic error message such as `Segmentation fault`. Our method for counting crash errors may over-count in the case where we are studying multiple versions of the same compiler, and several of these versions contain the same (unfixed) bug. However, because the symptom of this kind of bug typically changes across versions (e.g., the line number of an assertion failure changes due to surrounding code being modified), it is difficult to reliably avoid double-counting. We have not attempted to do so.

We tested each compiler using vanilla optimization options ranging from “no optimization” to “maximize speed” and “minimize size.” For example, GCC and LLVM/Clang were tested using `-O0`, `-O1`, `-O2`, `-Os`, and `-O3`. We did not use any of the architecture-specific (e.g., GCC’s `-m3dnow`) or feature-specific (e.g., Intel’s `-openmp`) options that typically make compilers extremely easy to crash.

We generated 1,000 unique swarm configurations, each of which selectively disables, with 50% probability, some of:

- declaration of `main()` with `argc` and `argv`

- the comma operator, as in $x = (y, 1)$;
- compound assignment operators, as in $x += y$;
- embedded assignments, as in $x = (y = 1)$;
- the auto-increment and auto-decrement operators $++$ and $--$
- goto
- integer division
- integer multiplication
- long long integers
- 64-bit math operations
- structs
- bitfields
- packed structs
- unions
- arrays
- pointers
- const-qualified objects
- volatile-qualified objects
- volatile-qualified pointers

3) *Results—Distinct Bugs Found:* The top-level result from this case study is that with all other factors being equal, a week of swarm testing on a single, reasonably fast machine found 104 distinct ways to crash our collection of compilers, whereas Csmith’s heavily tuned default configuration found only 73—an improvement of 42%. Table III breaks these results down by compiler. A total of 47,477 random programs were tested by the default Csmith run, of which 22,691 crashed at least one compiler.² A total of 66,699 random programs were tested by swarm, of which 15,851 crashed at least one compiler. Thus, swarm found 42% more distinct ways to crash a compiler while finding 30% fewer actual instances of crashes. Test throughput increased for the swarm case because simpler test cases (i.e., those lacking some features) are faster to generate and compile.

Of course, in any randomized experiment, flukes are possible. To test the statistical significance of our results, we split each of the two one-week tests into seven independent 24-hour periods and used the t-test to check if the samples are from different populations. The resulting p-value for these data is 0.00087, indicating significance at the 99.9% level. We also normalized for number of test cases, giving the non-swarm case a 40% advantage in terms of CPU time. Swarm remained superior in a statistically significant sense.

4) *Results—Code Coverage:* Table IV shows the effect that swarm testing has on coverage of two compilers: LLVM/Clang 2.9 and GCC 4.6.0. To compute confidence intervals for the increase in coverage, we ran seven 24-hour tests for each compiler and for each of swarm and the default configuration. The absolute values of these results should

²We realize that it may be hard to believe that nearly half of random test cases would crash some compiler. Nevertheless, this is the case. The bulk of the “easy” crashes come from Open64 and Sun CC, which have apparently not been the target of much random testing. Clang, GCC, and Intel CC are substantially more robust.

Table IV
COMPILER CODE COVERAGE USING THE DEFAULT CSMITH CONFIGURATION AND USING SWARM TESTING

Compiler	Metric	Default	Swarm	Change (95% conf.)
Clang	line	95,021	95,695	446–903
	branch	63,285	64,052	619–915
	function	43,098	43,213	37–193
GCC	line	142,422	144,347	1,547–2,303
	branch	114,709	116,664	1,631–2,377
	function	9,177	9,263	61–112

Table V
TOP TRIGGER AND SUPPRESSOR FEATURES IN SWARM TESTING OF C COMPILERS

Triggers		Suppressors	
Pointers	33%	Pointers	41%
Arrays	31%	Embedded assignments	24%
Structs	29%	Jumps	21%
Volatiles	21%	Arrays	17%
Bitfields	15%	$++$ and $--$	16%
Embedded assignments	15%	Volatiles	15%
Consts	13%	Unions	13%
Comma operator	11%	Comma operator	11%
Jumps	11%	Long long ints	11%
Unions	11%	Compound assignments	11%
Packed structs	10%	Bitfields	10%
Long long ints	10%	Consts	10%
64-bit math	10%	Volatile pointers	10%
Integer division	8%	64-bit math	8%
Compound assignments	8%	Structs	7%
Integer multiplication	6%	Packed structs	7%

Values in the table show the percentage of compiler crash bugs that were statistically likely to be triggered or suppressed by test cases containing the listed C program features.

be taken with a grain of salt: LLVM/Clang and GCC are both large (2.6 MLOC and 2.3 MLOC, respectively) and contain much code that is impossible to cover during our tests. Irrespective of the absolute coverage values, we believe that the incremental coverage values—for example, around 2,000 additional branches in GCC were covered—support our claim that swarm testing provides a useful amount of additional test coverage.

5) *Results—Significant Features:* During the week-long swarm test run described in Section III-B3, swarm testing found 104 distinct ways to crash a compiler in our test set. Of these 104 crash symptoms, 54 occurred enough times for us to analyze the triggering test cases for significant features. (Four occurrences are required for a feature present in all four test cases to become recognizable as not including the baseline 50% occurrence rate for the feature within its 95% confidence interval.) 52 of these 54 crashes had at least one feature whose presence was significant and 42 of them had at least one feature whose absence was significant.

Table V shows which of the C program features that we used in this swarm test run were statistically likely to trigger or suppress compiler crash bugs. Some of these data, such as the frequency with which pointers, arrays, and structs

```

struct S1 {
    int f0;
    char f1;
} __attribute__((packed));

struct S2 {
    char f0;
    struct S1 f1;
    int f2;
};

struct S2 g = { 1, { 2, 3 }, 4 };

int foo (void) {
    return g.f0;
}

```

Figure 5. Code triggering a crash bug in Clang 2.6

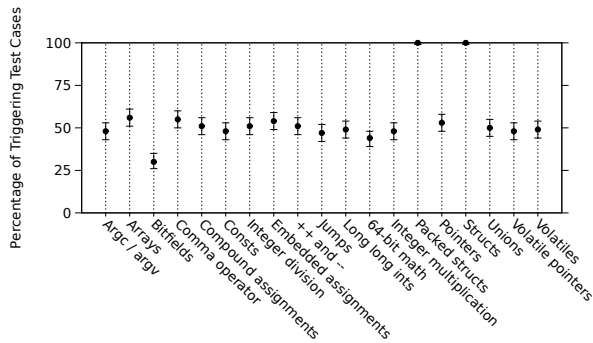


Figure 6. 95% confidence intervals for the percentage of test cases triggering the Clang bug triggered by the code in Figure 5 containing each Csmith program feature.

trigger compiler bugs, are unsurprising. On the other hand, we certainly did not expect pointers, embedded assignments, jumps, arrays, or the auto increment/decrement operators to figure so highly in the list of bug suppressors.

We take two lessons from the data in Table V. First, some features (most notably, pointers) strongly trigger some bugs while strongly suppressing others. This observation directly motivates swarm testing. Second, our intuitions (built up over the course of reporting 400 compiler bugs) did not serve us well in predicting which features would most often trigger and suppress bugs.

6) *An Example Bug:* A bug we found in Clang 2.6 causes it to crash when compiling—at any optimization level—the code in Figure 5, with this error message:

```

Assertion 'NextFieldOffsetInBytes <= FieldOffsetInBytes &&
"Field offset mismatch!'" failed.

```

This crash happened 395 times during our test run. Figure 6 shows that packed structs (and therefore, of course, also structs) are found in 100% of test cases triggering this bug. This is not surprising since the error is in Clang’s logic for dealing with packed structs. The only other feature that has a significant effect on the incidence of this bug is bitfields, which suppresses it. We looked at the relevant Clang source code and found that the assertion violation happens in a function that helps build up a struct by adding

its next field. It turns out that this (incorrect) function is not called when a struct containing bitfield members is being built. This explains why the presence of bitfields makes it less likely that this bug will be triggered.

IV. RELATED WORK

Swarm testing is a low-cost and effective approach to increasing the diversity of randomly generated test cases. It is inspired by swarm verification [12], which runs a model checker in multiple search configurations to improve the coverage of large state spaces. The core idea of swarm verification is that given a fixed budget of time and memory, a “swarm” of diversely defined searches can explore more of the state space than a single “optimal” search configuration. Swarm verification is successful in part because a single “best” search configuration cannot easily exploit parallelism: the communication overhead for checking whether states have already been visited by another worker gives a set of independent searches an advantage. This advantage disappears in a trivially parallelizable automated testing approach such as random testing: random testing runs are completely independent, with no need to communicate results or share storage. The benefits of swarm testing thus do not depend on any loss of parallelism inherent in the “monolithic” approach to configuration, or, like the parallel randomized state-space search proposed by Dwyer et al. [13], on the failure of depth-first search to “escape” a subtree when exploring a very large state space. Rather, the results completely reflect the value of (feature omission) diversity in test configurations.

Swarm verification and swarm testing are orthogonal approaches in that swarm verification could be applied in combination with feature omission to produce further search diversity. Test-case diversity is a motivating factor, even when not explicitly acknowledged as such, in a wide variety of software testing approaches [14]. Approaches to diversity are themselves quite diverse, differing both in how they define diversity and in how they seek diverse test cases.

Partition testing [15], combinatorial testing [16], and adaptive random testing [17] seek diversity at the input level. They are therefore somewhat similar to swarm testing, which uses diversity in the input configurations of a random test-case generator.

Partition testing is coverage-based: the input space is divided into disjoint partitions, and the goal of testing is to sample (cover) at least one input from each partition. Two underlying assumptions are that (1) the partition forces diversity and (2) inputs from a single partition are homogeneous and largely interchangeable. Unfortunately, without a sound basis for the partitioning scheme (which is seldom possible), partition testing often performs worse than pure random testing [15] in terms of real behavioral diversity. Category-partition testing [18] may improve partition testing, but requires a formal specification and considerable machinery and effort to achieve. Swarm testing differs

from partition testing in that it does not rely on the same assumptions. Although configurations used in swarm testing can be thought of as partitions, (1) the partitions are not required to be disjoint and (2) there is no assumption of homogeneous behavior within a partition. Swarm testing uses its “partitions” purely to increase input diversity, rather than relying on any equivalence classes between inputs.

Combinatorial testing seeks to test features in different combinations, toward the goal of maximizing test coverage or reducing total testing time. Swarm testing with 50% probabilities on configuration features will somewhat resemble combinatorial testing, in that some tests with all pairs of features are expected to be executed in most cases. In swarm testing, however, there is no assumption about how features interact, nor is there an attempt to be exhaustive, and the emphasis is on feature omission diversity, not on attempting to cover positive feature interactions.

Adaptive random testing modifies traditional random testing by sampling the space of tests and only executing the test most “distant,” as determined by a distance metric over inputs, from all previously executed test cases. Many variations on this approach have been proposed, all essentially retaining the emphasis on diversity of inputs. Unlike adaptive random testing, swarm testing does not require the work—either human effort or run-time overhead—of computing a distance metric, and therefore has lightweight requirements on the test engineer. The kind of “feature breakdown” that swarm requires is commonly provided by test generators; in our experience developing over a dozen random-testing and model-checking test harnesses, we implemented the generator-configuration options long before we considered utilizing them as part any methodology other than simply finding a good default configuration. Swarm testing has been applied to real-world systems with extremely encouraging results; ART has not been shown to be effective for complex real-world programs [19].

Structural testing [20], statistical testing [21], many meta-heuristic testing approaches [22], and even concolic testing [23] can be viewed as aiming at a set of test cases exhibiting diversity in the targets they cover—e.g., statements, branches, or paths. Other approaches make the goal of maximum behavior diversity even more explicit, as in testing via operational abstractions [24] and for contract violations [25]. Feedback-directed testing [6] similarly attempts to diversify objects created during testing. A primary difference between these techniques and swarm testing is that they require the extraction of coverage information, behavioral invariants, or other information. In addition, they often require expensive symbolic execution or algorithmic learning. Such analyses are often difficult to apply without source-code access, and they sometimes fail to scale to complex real-world programs. More scalable approaches [26] tend to be focused on a certain kind of application and certain types of faults (e.g., security vulnerabilities). Another difference is that many of

these approaches iteratively refine their notion of diversity in such a way that future exploration relies on past results, making them nontrivial to parallelize. Swarm testing, in contrast, is inherently embarrassingly parallel.

V. CONCLUSIONS AND FUTURE WORK

Swarm testing relies on the following claim: for realistic systems, *randomly excluding some features from some tests* can improve coverage and fault detection, compared to a test suite that potentially uses every feature in every test. The benefit of using of a single inclusive default configuration—that every test can potentially expose any fault and cover any behavior, heretofore usually taken for granted in random testing—does not, in practice, make up for the fact that *some features can, statistically, suppress behaviors*. Effective testing therefore may require feature omission diversity. We show that this not only holds for simple container-class examples (e.g., pop operations suppress stack overflow) but for a widely used flash file system and 17 production-quality C compilers. For these real-world systems, if we compare testing with a single inclusive default configuration to testing with a set of 100–1,000 unique configurations, each omitting features with 50% probability per feature, we have observed (1) significantly better fault detection, (2) significantly better branch and statement coverage, and (3) strictly superior mutant detection. Configuration diversity does indeed produce better testing in many realistic situations.

Swarm testing was inspired by swarm verification, and we hope that its ideas can be ported back to verification problems. We also plan to investigate applying swarm ideas in the context of bounded exhaustive testing and learning-based testing methods. Finally, we believe there is room to better understand *why* swarm provides its benefits, particularly in the context of large, idiosyncratic SUTs such as compilers, virtual machines, and OS kernels. More case studies will be needed to generate data to support this work. We also plan to investigate how swarm testing’s increased diversity of code coverage in test cases can benefit fault localization and program understanding algorithms relying on statistical analysis of large test suites [27]; traditional random tests are far more homogenous than swarm tests.

We have made Python scripts supporting swarm testing available at <http://beaversource.oregonstate.edu/projects/cswarm/browser/release>.

REFERENCES

- [1] R. Hamlet, “Random testing,” in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.
- [2] J. Andrews, F. Li, and T. Menzies, “Nighthawk: A two-level genetic-random unit test data generator,” in *Automated Software Engineering*, 2007, pp. 144–153.
- [3] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, “Random test run length and effectiveness,” in *Automated Software Engineering*, 2008, pp. 19–28.

- [4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.
- [5] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.
- [6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.
- [7] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *International Symposium on Software Testing and Analysis*, 2010, pp. 219–230.
- [8] "Yaffs: A flash file system for embedded use," <http://www.yaffs.net/>.
- [9] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.
- [10] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28(2), pp. 183–200, 2002.
- [11] A. Groce, "(Quickly) testing the tester via path coverage," in *Workshop on Dynamic Analysis*, 2009.
- [12] G. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2010.
- [13] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *International Conference on Software Engineering*, 2007, pp. 3–12.
- [14] T. Y. Chen, "Fundamentals of test case selection: diversity, diversity, diversity," in *International Conference on Software Engineering and Data Mining*, 2010, pp. 723–724.
- [15] R. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, 1990.
- [16] D. R. Kuhn, D. R. Wallace, and J. Albert M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.
- [17] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Advances in Computer Science*, 2004, pp. 320–329.
- [18] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.
- [19] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness," in *International Symposium on Software Testing and Analysis*, 2011, pp. 265–275.
- [20] P. Thevenod-Fosse and H. Waeselyncx, "An investigation of statistical software testing," *J. Software Testing, Verification, and Reliability*, vol. 1, no. 2, pp. 5–26, 1991.
- [21] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering*, vol. 36, pp. 763–777, 2010.
- [22] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, pp. 742–762, 2010.
- [23] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Programming Language Design and Implementation*, 2005, pp. 213–223.
- [24] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *International Conference on Software Engineering*, 2003, pp. 60–71.
- [25] Y. Wei, Y. Pei, H. Roth, A. Horton, M. Steindorfer, C. A. Furia, M. Nordio, and B. Meyer, "Stateful testing: Finding more errors in code and contracts," *Computing Research Repository*, August 2011.
- [26] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008.
- [27] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.