Help, Help, I'm Being Suppressed! The Significance of Suppressors in Software Testing

Alex Groce Chaoqiang Zhang Mohammad Amin Alipour
School of Electrical Engineering and Computer Science
Oregon State University
{grocea, zhangch, alipourm}@onid.orst.edu

Eric Eide Yang Chen John Regehr
School of Computing
University of Utah
{eeide, chenyang, regehr}@cs.utah.edu

***Abstract*—Test features are basic compositional units used to describe what a test does (and does not) involve. For example, in API-based testing, the most obvious features are function calls; in grammar-based testing, the obvious features are the elements of the grammar. The relationship between features as abstractions of tests and produced behaviors of the tested program is surprisingly poorly understood. This paper shows how large-scale random testing modified to use diverse feature sets can uncover *causal* relationships between what a test *contains* and what the program being tested *does*. We introduce a general notion of observable behaviors as *targets*, where a target can be a detected fault, an executed branch or statement, or a complex coverage entity such as a state, predicate-valuation, or program path. While it is obvious that targets have triggers — features without which they cannot be hit by a test — the notion of *suppressors* — features which make a test *less* likely to hit a target — has received little attention despite having important implications for automated test generation and program understanding. For a set of subjects including C compilers, a flash file system, and JavaScript engines, we show that suppression is both common and important.**

I. Introduction

This paper uses large-scale random testing to show that what is *not* included in a test can frequently be a key factor in determining its effectiveness for coverage and fault detection. Large-scale random testing (using swarm testing [1]) provides a "statistical window" into the relationship between what we can *control* about a test case (e.g., which APIs are called, which features of the C or JavaScript language are used, which HTML tags appear, etc.) and what we *observe* when we execute the test case (e.g., coverage of statements/branches/paths, faults detected, invariants that hold, program states reached, etc.). The relationship between controllable and observable aspects of testing provides insights into the structure of a program's state space and how to generate more effective test suites. In particular, *omitting* features from tests can *positively* affect the ability of a test to explore interesting behavior.

A. Features: Controllables

A *feature* in this paper means a property of a *test case for a system* [1]. This use is different than (but related to) features in customizable programs/software product lines [2]. Formally, a feature is any property of a test case that can be *imposed* at test generation time without increasing computational complexity. For example, when testing an API-based system, every function/method can be considered a feature — it is simple to generate tests that contain or do not contain a given call. For grammar-based test generation, a feature might be a terminal or production in the grammar — e.g., when testing a C compiler, including or not including pointer variables. Features should be not only easy to impose at generation, but at worst linear to detect in a test case. A feature set is a constraint on a test case, in terms of what it must not contain.[1]

```
yaffs_symlink("/ram2k", "/ram2k/fssr");
int fd0 = yaffs_open("/ram2k/fssr/yr", O_TRUNC|O_WRONLY);
yaffs_lstat("/ram2k/fssr/sshd", &st_buff);
```

(A) Simplified random test case for the YAFFS2 file system, with boilerplate code removed. Features are API calls. The feature set is: {`symlink`, `lstat`, `open`}. Many features (`close`, `write`, etc.) are not present.

```
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 shell. Features here include labels, assignments, and `with` blocks, but do not include `try` blocks, infinite loops, or XML.

Fig. 1. Features for Random Test Cases

This concept of features is intuitive and naturally adopted by the developers of (random) testing systems. Most industrial-strength random testing systems (and many model checking harnesses) already support this notion of features [3], [4], [5], [6]. Figure 1 shows test cases generated by two such systems and discusses their feature sets. It is also quite easy to impose feature sets in a symbolic or concolic testing harness. In this paper we use "feature set" and "configuration" interchangeably, as a feature set is simply a configuration for a test generator that determines which tests can be produced.

B. Targets: Observables

A *target* is a behavior of a software system that some test cases may produce and other tests may not produce. The most obvious targets are faults and coverage entities. A test case may either expose a given fault or not, execute a given block or not, take a branch or not, execute a given path or not, reach a state in a Boolean abstraction or not, and kill a given mutant or not. Faults, blocks, branches, paths, predicate-complete test (PCT) coverage points [7], and mutants therefore are all targets, and a test case *hits* a target if it exposes/covers it. It is tempting to

[1]Most random testing systems do not guarantee that an allowed feature is actually present, but non-inclusion is usually very rare.

equate targets with sets of program states — certainly sets of program states are targets (targets thus include all reachability properties). However, some targets (e.g., path coverage) may not be expressible as reachability properties without the use of history variables. For the examples in Figure 1, the hit targets include all statements and branches executed when performing the file system operations or running the JavaScript shell (and a crash, for Mozilla's `js` version 1.6). Targets are the atomic building blocks of the things we hope to "get out" of testing.

C. *Triggers and Suppressors*

Given the concepts of features and targets, we can ask whether a feature f "helps" us 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 *requires* the call to be in the test case. Less obviously, features may make it *harder* to hit some targets. 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 [1]. 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.

No one questions the importance of triggers in testing. That tests must, e.g., make certain calls or include certain strings to cause behavior is self-evident. Attempts to generate behavior, whether by producing very large numbers of tests with random generation or by symbolic execution, essentially define the field of automated testing, and motivate most manual testing. However, the idea that certain properties of tests can (even if the right triggers are present) make a test *less likely* to explore some behaviors has seldom been explored. In previous work [1], we suggested suppression as one explanation for the effectiveness of "swarm testing," a variation/improvement of random testing that omits features at random from configurations. The primary contribution of this paper (Sections III–IV) is an empirical demonstration that *suppressors are extremely common*, both in the sense that targets frequently have suppressors and that many features act as suppressors for some targets. For some subjects, suppression was observed more often than triggering, and the strength of suppressors was sufficiently high (though lower than the strength of triggers) to produce a major effect on test effectiveness. A secondary contribution (Sections IV-E–VI) is a discussion of causes of suppression, the value of suppression in program understanding and test design, and the impact of suppression on automated testing methods, including examination of possible mitigation strategies.

II. Formally Defining Suppression

A. *Estimating Hitting Fractions*

Given a population of test cases with varying feature sets hitting a varying set of targets, we want to compute the fraction of tests hitting a target t that contain a given feature f. If tests hitting t never contain f, we can conclude (if our tests are drawn from a distribution in which f is often enabled) that f likely disables t — it is a very strong suppressor. Conversely, if *all* tests hitting t contain f, it is likely *necessary* to trigger t. Unfortunately, we cannot exhaustively test realistic systems. However, statistical methods exist for estimating such proportions, given statistically independent tests. The Wilson score [8] is a frequently used method (a binomial proportion confidence interval) for estimating such proportions. Wilson scores work well with sampling error even for small populations and extreme probabilities. Using Wilson scores and sampled tests, we can determine if f is a suppressor or a trigger for t with any desired confidence C (e.g., $C = 95\%$) as follows.

Given feature f, target t, and test case population P where f appears in tests at rate r, compute a C confidence Wilson score interval (l,h) on the true proportion of t-hitting tests that contain f. If $h < r$, we can be $C\%$ confident that f suppresses t. The lower h is, the more suppressing f is for t. Conversely, when $l > r$, f is a trigger for t. If neither of these cases holds, we can say that f is irrelevant to t. We can use the appropriate bound (lower or upper) as a conservative estimate for the true fraction F of t-hitting tests containing f:

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

F is easily interpreted when the rates for features are set at 50% in P. If n tests hit t in P, then there should be $n \cdot \frac{F}{0.5}$ tests hitting t in a population P_f of the same size where every test includes f. F of 1.0 predicts that hits will be *twice as common* over tests always containing the feature, and F of 0.0 predicts that tests always containing the feature will *never* hit t. Measuring F allows us not only to determine if a feature is a suppressor or trigger, but allows us to predict its quantitative effect on testing for a given target.

B. *Features and Causality*

A key point to note is that F typically describes *causal* relationships. In many statistical analyses, it is only possible to show correlations, not causality, even if statistical confidence is complete. If c is anti-correlated with e, it may simply be because some unknown third effect c' causes both the presence of c and the absence of e: e.g., large numbers of typos in comments in a source code section could be correlated strongly with the absence of correct error handling in the code. The relationship is possibly useful (if it were strong enough, we might devote more static analysis or testing effort at portions of code with such comments) but it is not *causal*. It would be unreasonable to propose "spellcheck your comments" as a method for reducing program faults! We believe that "carelessness" or "hurry" or some other factor causes both effects, rather than that errors in comments actually cause faults in source code. Features, however, are (by definition) *controllable*. If the presence of f decreases the presence of t, statistically, f *has no other causes* and therefore f and t cannot both be the result of some unknown primary cause. It is crucial to note that this causality is in the full context of the system tested *and* a test population/generation method.

Full controllability significantly increases the value of examining trigger/suppressor relationships by random sampling: relationships found are generally going to be *causal truths* about the structure of the software's execution space when sample sizes are reasonably large. It is critical to note that this does not extend to examining relationships *between* triggers. If we observe correlation between, e.g., coverage of statement S_1 and statement S_2, this may obviously be the result of a shared cause. It is only the fact that features are true controllables, rather than observables, that makes our results causally meaningful.

There is one subtle issue related to this analysis. Most of our features are completely independent, with no interactions. However, in a few cases feature f_1 requires another feature f_2 in order to actually express, even if f_1 is enabled: e.g., generating C programs with bitfields (f_1) is impossible without generating structs (f_2). We believe that this simple kind of dependency is harmless. The key point is that the presence/absence of f_1 in tests without f_2, hitting any target t, is completely random. The rate r for f_1 is artificially high in F computations, and l and h values are potentially artificially high as well, but F is based on *relative* values. The primary "bias" is simply that F will normalize results to match those in a sample where r was, in fact, equal to its artificially inflated value. For targets with very few hits, however (a very small portion of our targets), we could occasionally see false *triggering* (but not false *suppression*), or miss a suppression effect. More complex and problematic forms of dependency, e.g., where f_2 always disables f_1, were not present in our subjects.

C. Research Questions

While the basic concept of suppression is intuitively clear, little is known about its frequency or importance in testing. The key questions investigated in this paper concern the frequency and degree of suppression in software systems:

- **RQ1:** How many features suppress some targets?
- **RQ2:** How many targets are suppressed by some feature?
- **RQ3:** How many suppressors do targets have?
- **RQ4:** How strong is the effect of suppression?

III. Experimental Methodology

The only practical way to answer these questions is to apply large-scale random testing to real software systems. Estimates of $F(f,t)$ are based on 95% confidence level Wilson score intervals over P produced by realistic random testing systems. Unlike model checking, adaptive random testing, or random testing with inter-test feedback [5], traditional random testing satisfies the condition of statistical independence between tests. However, it usually includes *all* features in each test of nontrivial length, with very high probability, which makes r too high to estimate F effectively. Our results are therefore based on swarm testing [1], in which each generated test uses a configuration that *randomly omits features* with probability 0.5 for each f. Swarm testing has the further advantage that it may be more effective than traditional random testing, which increases the set of targets. $F(f,t)$ was only computed for targets hit at least 4 times, since 95% confidence intervals always include the base rate for smaller samples. We measured basic code coverage with blocks in some cases and statements in others due to tool availability; blocks were preferred when easily computable, as statements include many targets that form an equivalence class.

TABLE I
EXPERIMENTAL SUBJECTS

	Features			
SUT	#	Type	LOC	# Tests
YAFFS2	37	API calls	$\sim$15K	5K
C Compilers	28	C language	$>\sim$400K	$\sim$200K
GCC snapshot	21	C language	$\sim$500K	5K
Mozilla js 1.6, 1.7	266	JavaScript	$\sim$42K	$\sim$100K
Mozilla js 1.8.5	266	JavaScript	$\sim$42K	$\sim$20K
Lobo HTML parser	100	HTML Tokens	$\sim$975	5K
SGLIB rbtree	7	API calls	476	12.7K
Java containers	2	API calls	$<$ 600	$\sim$5K

A. Experimental Subjects

Table I summarizes key details of our experimental subjects: the number and type of features, approximate LOC, and the number of tests used to produce statistics.

1) YAFFS2: The YAFFS2 flash file system [9] was the image file system for early versions of Android. YAFFS2 tests can include or exclude any of 37 core API calls. Feedback [4], [5] in the YAFFS2 tester ensures that calls such as `close` and `readdir` occur only in states where valid objects are available. The version of YAFFS2 used is $\sim$15 KLOC of C code. YAFFS2 experiments used a variety of targets, made possible by the modest size of the code: basic blocks, branches, mutants, and even automatically instrumented predicate-complete test coverage states [7], a very fine-grained coverage measure. Mutation analysis was based on 1,000 randomly sampled mutants generated by the software of Andrews et al. [10]. Random sampling of mutants has been shown to provide useful results approximating fault detection in cases where using all mutants is not practical [11]. (YAFFS2 has well over 14,000 possible mutants.) YAFFS2 results are based on 5,000 tests of length 200 API calls (10^6 file system operations, with an additional 10^9 operations required to evaluate mutants, motivating the relatively small number of tests).

2) C Compilers: Csmith [3] uses a customized grammar for a subset of the C language to generate random C programs. The major challenge is to avoid generating invalid code; Csmith accomplishes this by interleaving generation with static analysis of the partially generated program. The resulting code — at least partially because it contains infinite loops, irreducible flow graphs, and other constructs that C compilers do not routinely encounter — tends to be an effective stress-testing tool for optimizing compilers [3].

We used programs generated by Csmith to test a collection of production-quality C compilers including GCC 3.2.0, 3.3.0, 3.4.0, 4.0.0, 4.1.0, 4.2.0, 4.3.0, 4.4.0, 4.5.0, 4.6.0, and LLVM/Clang 2.6, 2.7, 2.8, 2.9. All of these compilers were configured to generate x86-64 code and were invoked at standard optimization levels (-O0, -O1, -O2, -Os, and -O3). Results are based on generating and compiling about 200,000 test cases using Csmith and each of the 14 compilers at each of the optimization levels. Code generated by the compilers was not executed, as it is difficult to automatically cluster wrong-code bugs; results are based on whether the compilers generated code at all, as opposed to terminating in an abnormal fashion. During testing, we found 56 unique compiler-crash bug symptoms. A "unique symptom" is represented by a particular string that a compiler emits while crashing. Real faults are the most interesting and important targets in testing. The compiler results below combine crashes from all compilers to form a target set, in order to provide a large enough set of targets to provide answers to the research questions and because a tool like Csmith is intended to work as a general-purpose compiler testing tool. Results over all compilers are more likely to apply to future testing than results for the small number of crashes detected in any one compiler. The trends observed over all compilers held for individual compilers as well.

A second compiler experiment used Csmith to generate tests for the August 8, 2012 developer snapshot of GCC, and used statements as targets. The source contained $\sim$550 KLOC, and tests covered about $\sim$125K. Because coverage measurement reduces test throughput and the extremely large number of targets increases analysis time, these experiments cover only 21 features (removing some that cause slowdowns in practical testing) and 5,000 test cases.

3) JavaScript Shells: The `jsfunfuzz` tool [12] has been used to find over 1,800 bugs in Firefox's JavaScript engine, many of which were exploitable memory-safety bugs. Tests in this case are sequences of JavaScript language constructs, randomly generated by a highly complex set of recursive functions tuned to cover combinations of language features; `jsfunfuzz` not only executes these constructs, but checks for basic semantic properties (e.g., round-trip properties). Subjects were the SpiderMonkey `js` JavaScript shell versions 1.6, 1.7, and 1.8.5 (the latest source release), all C/C++ programs of $\sim$120 KLOC, including comments, of which $\sim$42,000 are executable statements. For versions 1.6 and 1.7 approximately 100,000 tests containing 1,000 JavaScript shell inputs were generated. Features were automatically derived from the full set of randomized case splits in the `jsfunfuzz` code generation (production rules in the grammar), yielding a set of 266 features, using a short ($<$ 50 lines of Python) "swarm fuzzer" that produces versions of `jsfunfuzz` with different random features. Testing version 1.6 revealed 37 distinct failure symptoms with at least 4 instances, including several showing a serious memory error, a wide variety of JavaScript semantics violations, one case of non-termination, and two distinct out-of-memory errors. Testing version 1.7 produced 52 different distinct failure symptoms (but only one variety of segmentation fault and memory exhaustion), though some of these were similar semantic issues that may result from a single underlying fault. For version 1.8.5, we gathered statement coverage for each test with gcov, which greatly reduced the throughput of testing; the time to execute approximately 20,000 tests was much larger than for 100,000 runs of 1.6 and 1.7. Only 13 distinct crashes (including one memory failure and no segmentation faults) were detected for version 1.8.5, and each only occurred once in over 20,000 tests. We note in passing that for JavaScript, swarm tests had failure rates and total distinct-failures-found values almost twice those of tests using a single configuration (not used in our analysis), confirming the value of swarm testing and supporting our hope to reach "hard to hit" targets.

4) HTML Parser: We tested the HTML parser included in the Lobo project [13], a pure-Java web browser. The parser code itself (the testing target) is about 975 lines of code, supported by an HTML document class and other infrastructure. Coverage was computed over the HTML parser code only. The features in this experiment were 100 tokens including most common HTML tags. Targets in this case were branches and statements. Results were based on 5,000 tests of length 50.

5) SGLIB rbtree: SGLIB is a popular library for data structures in C [14]; its red-black tree implementation is an example of a relatively simple, but nontrivial (i.e., it has semi-redundant API calls, etc.), container. The code is 476 LOC in C. Basic blocks, branches, and (all) mutants served as targets. Results are based on 12,700 tests of length 10.

6) Java Container Classes: The final set of subjects are taken from Java container classes used in multiple papers to compare test generation methods [15], [16]. These all have very small APIs (3 methods — `insert`, `remove`, and `find`). For these subjects, the only features are `remove` and `find`, as tests with no `insert` operations are essentially identical (as state never changes). One possible explanation of the previous inattention to suppression effects may be that suppression is less problematic for simple container classes, which have been the subjects of many studies [15], [16], [17]. These subjects were easily covered by $\sim$5,000 length-200 tests each.

B. Threats to Validity

The primary threat to validity is that we only consider a modestly sized set of subjects. However, these systems are realistic and relevant: automated testing is most desirable for critical systems code, and reusable container classes are the most popular subjects in the literature. Moreover, all of the nontrivial test generators were developed and used prior to this study. For gaining insights into automated testing of critical software, test cases produced by realistic test generators focused on finding faults seems like the best population to study.

Results over features are largely "correct" for these subjects, with the caveat that dependencies may produce a few spurious triggers and hide some suppressors. Results over targets, however, *could* be artifacts of which targets our test population was able to cover. If a branch was never covered, or a compiler never crashed in a particular way, we cannot draw any conclusions about how features influence the rate of detection for that branch/crash, but can only place a bound on the probability of hitting that target with the test population. In general, target coverage is undecidable, so complete results over all targets is not a realistic goal. Test suites are large enough in all cases, however, to cover targets that are within the domain of random or systematic automated testing, but

TABLE II
ROLES FILLED, BY FEATURE

I=Irrelevant, S=Suppressor, T=Trigger, C=Crashes
M=Mutants, BL=Blocks, BR=Branches, ST=Statements

SUT	Targ.	IST	IS	IT	ST	I	S	T
YAFFS2	M	29	1	7	-	-	-	-
YAFFS2	BL	31	-	5	-	1	-	-
YAFFS2	BR	33	-	3	-	1	-	-
YAFFS2	PC	37	-	-	-	-	-	-
Compilers	C	22	2	4	-	-	-	
gcc	ST	21	-	-	-	-	-	-
js 1.6	C	139	59	47	-	21	-	
js 1.7	C	178	40	41	-	7	-	
js 1.8.5	ST	266	-	-	-	-	-	-
HTML	ST	28	21	21	-	30	-	-
HTML	BR	30	27	21	-	22	-	-
rbtree	M	4	1	2	-	-	-	-
rbtree	BL	5	-	2	-	-	-	-
rbtree	BR	4	1	2	-	-	-	-
AVLTree	ST	1	1	-	-	-	-	-
AVLTree	BR	1	1	-	-	-	-	-
FibHeap	ST	1	-	1	-	-	-	-
FibHeap	BR	1	-	1	-	-	-	-
HeapArr	ST	-	-	1	-	1	-	-
HeapArr	BR	-	-	1	-	1	-	-
TreeMap	ST	1	-	1	-	-	-	-
TreeMap	BR	1	-	1	-	-	-	-
Sum		833	154	161	-	84	-	-

have low probability of appearance. In previous work we have shown that swarm testing tends to be as or more effective than traditional random testing with a single configuration [1]: these results cover most targets likely to be found in random testing, except where large clusters of machines are applied for weeks of testing. One side-effect of this problem of unexplored targets is difficulty examining the relationship between ease of hitting a target and its suppressors. Because the testing randomly omits features, it is hard to hit targets that require large numbers of features in combination, as few such tests will be generated. The *low-probability* targets hit will therefore be biased towards those requiring only a few features, with suppressors. This problem is difficult to avoid. Concolic testing *might* hit such targets (though we are not sure it would work well for our more complex subjects), but would not tell us about the *population* of tests hitting the targets. Random testing with most features included in each test case could potentially hit more such targets, but makes detecting suppressors and triggers nearly impossible. Minimizing tests [18] cannot distinguish suppressors from irrelevant features.

IV. EXPERIMENTAL RESULTS

A. RQ1: Features Often Suppress Some Targets

As Table II shows, most features for most Subjects Under Test (SUTs) we examined were triggers for some targets but suppressors for other targets. The table shows, for each subject and target type, how many features served in all roles (for some target), each combination of two roles, or in only one role. In this table and all following tables, 'C' stands for Crashes and other failures, 'M' stands for Mutants, 'BL' stands for Blocks, 'BR' stands for Branches, 'PC' stands for predicate-complete test coverage [7], and 'ST' stands for Statements. 'I' stands for Irrelevant, 'S' stands for Suppressor, and 'T' stands for Trigger. Column headings in Tables II and III indicate the range of roles that a feature fills (or a target has some feature filling). For example, 'IST' indicates that a feature is irrelevant to at least one target, a suppressor for at least one target, and a trigger for at least one target. 'IS' indicates the same, except that the feature is never a trigger. In this and following tables, the dividing line separates results over more complex systems with at least 7 API calls or grammar elements from simple systems with few API calls and no overlap between features.

For the complex systems software, the range of roles taken by features was considerable: 4 compiler features (`paranoid`, `union-read-type-sensitive`, `consts`, and `return-unions`) were never suppressors, and 2 compiler features (`muls` and `arg-structs`) were never triggers. The other 22 compiler features served in all three roles for some crash. For the YAFFS2 tests, approximately 30 features served in all three roles for mutant, block, and branch targets. Calling `fstat` was only a suppressor or irrelevant for mutants, and 3–7 features were only irrelevant or triggering for each kind of target. The only feature that was never suppressing for any kind of target was `pwrite`. For version 1.6 of the JavaScript shell, *more* features were suppressors than triggers, and for 1.6 and 1.7 a large number of features were either irrelevant or suppressors only or irrelevant and triggers only.

For the HTML parser, there were many more purely irrelevant features but about $1/4$ of all features still served in all roles. For SGLIB-rbtree mutants and branches, 2 features were only triggers and 1 feature was only a suppressor. The other 4 features served in all roles. For blocks, as we would expect, *no* feature served as only a suppressor (given that the features are API calls, there are obviously blocks that can only be reached via that API call), so 5 features served in all roles. The only subject with no suppressors was a Java container class. Even for container classes, 50% of features served in all three roles for three SUTs. Overall, 987 of 1,232 features (where a feature is counted once for each SUT/target) were suppressing for some target, only slightly less than the 994 that were triggers. In terms of features, *suppression is roughly as common as triggering*. Only 84 features were always irrelevant.

B. RQ2: Many Targets Have Suppressors

Table III is similar in structure to Table II, except that here the values in the table indicate how many targets had some feature filling each role — that is, the 'IST' column here indicates the number of *targets* that had at least one suppressor, at least one trigger, and at least one irrelevant feature. Again, across all SUTs and target types, suppression was very common. *No* JavaScript crashes lacked suppressors. For gcc, almost 8,000 statements had suppressors but no triggers. The very large number of 'I' only targets for `js` 1.8.5, gcc, and YAFFS PCT coverage are due to initialization, etc.: statements/points covered by all (or almost all) tests. In general, triggering was at most twice as common as suppression.

TABLE III
ROLES FILLED, BY TARGET

SUT	Tar.	IST	IS	IT	ST	I	S	T
YAFFS2	M	95	-	99	-	38	-	-
YAFFS2	BL	328	-	670	-	455	-	-
YAFFS2	BR	432	-	755	-	451	-	-
YAFFS2	PC	7678	-	9990	-	6070	-	-
Compilers	C	49	1	5	-	1	-	-
gcc	ST	12841	7752	11973	702	81842	1813	8711
js 1.6	C	37	-	-	-	-	-	-
js 1.7	C	52	-	-	-	-	-	-
js 1.8.5	ST	7553	25	4185	-	9660	-	-
HTML	ST	226	-	84	-	37	-	-
HTML	BR	124	-	46	-	12	-	-
rbtree	M	17	-	15	109	3	-	-
rbtree	BL	34	-	35	116	-	-	-
rbtree	BR	44	-	29	140	1	-	-
AVLTree	ST	-	3	54	26	128	-	-
AVLTree	BR	-	3	23	23	60	-	-
FibHeap	ST	-	-	3	41	17	-	97
FibHeap	BR	-	-	2	12	6	-	13
HeapArr	ST	-	-	40	-	46	-	-
HeapArr	BR	-	-	12	-	9	-	-
TreeMap	ST	-	-	117	48	117	1	12
TreeMap	BR	-	-	39	13	36	-	4

TABLE IV
AVERAGE FEATURES IN EACH ROLE, PER TARGET

SUT	Tar.	Avg. I	Avg. S	Avg. T	S/T Ratio
YAFFS2	M	32.7	1.1	3.0	0.37
YAFFS2	BL	33.6	1.2	2.1	0.57
YAFFS2	BR	33.3	1.5	2.3	0.65
YAFFS2	PC	33.6	1.0	2.3	0.43
Compilers	C	22.2	2.1	3.2	0.65
gcc	ST	16.5	1.1	3.4	0.32
js 1.6	C	238.4	10.8	9.8	1.1
js 1.7	C	240.2	9.4	11.4	0.82
js 1.8.5	ST	256.6	2.9	5.4	0.54
HTML	ST	94.4	1.9	3.4	0.56
HTML	BR	93.7	2.1	3.6	0.58
rbtree	M	0.8	3.0	2.8	1.07
rbtree	BL	1.3	2.8	2.8	1.0
rbtree	BR	1.2	3.0	2.8	1.07
AVLTree	ST	1.5	0.1	0.4	0.25
AVLTree	BR	1.3	0.2	0.4	0.25
FibHeap	ST	0.2	0.3	1.5	0.2
FibHeap	BR	0.4	0.4	1.2	0.33
HeapArr	ST	1.5	0.0	0.5	-
HeapArr	BR	1.4	0.0	0.6	-
TreeMap	ST	1.2	0.2	0.6	0.33
TreeMap	BR	1.2	0.1	0.6	0.17

C. RQ3: Targets on Average Have a Small Non-Zero Number of Suppressors and a Small, Non-Zero Number of Triggers

Table IV shows the average number of features filling each role for each target, for each SUT and target type. The average number of irrelevant features varies quite simply, with the number of features. (The standard deviation of the average number of irrelevant features is approximately the standard deviation in the total number of features.) The ranges and variance for suppressors and triggers, however, are considerably smaller: JavaScript crashes had almost ten times as many features as compiler crashes, but had only ∼5 times as many average suppressors and ∼3–4 times as many triggers. This bounding is even more marked for the HTML parser. As the number of features increases, the number of suppressors and triggers per target may grow slowly, or even be bounded for many natural feature sets. This makes intuitive sense: as programs grow more complex, modularity somewhat limits interaction between features. Most code is not concerned with most features; it tends to only read and write a projection of the state space. (In a sense, this is why abstraction "works" in verification.) The SGLIB example suggests that systems above a certain minimal threshold may also have a lower bound on suppression and triggering: despite having only 7 total features, the average S/T numbers for SGLIB were in the same range as other subjects. The only subjects averaging less than one suppressor per target were the simple Java container classes, which may explain why testing of these subjects works well despite ignoring suppression.

TABLE V
STRENGTH OF SUPPRESSORS AND TRIGGERS

		S			T		
		Min.	Avg. Min.		Max.	Avg. Max.	
SUT	Tar.		feat.	tar.		feat.	tar.
YAFFS2	M	0.21	0.44	0.44	1.0	0.85	0.93
YAFFS2	BL	0.11	0.45	0.42	1.0	0.98	0.94
YAFFS2	BR	0.11	0.43	0.42	1.0	0.96	0.94
YAFFS2	PC	0.06	0.39	0.41	1.0	0.97	0.92
Compilers	C	0.08	0.41	0.36	1.0	0.76	0.81
gcc	ST	0.01	0.21	0.38	1.0	0.99	0.72
js 1.6	C	0.18	0.46	0.41	0.98	0.54	0.71
js 1.7	C	0.30	0.46	0.42	0.99	0.55	0.69
js 1.8.5	ST	0.04	0.38	0.44	1.0	0.77	0.71
HTML	ST	0.18	0.48	0.42	1.0	0.58	0.90
HTML	BR	0.18	0.48	0.42	1.0	0.57	0.89
rbtree	M	0.27	0.37	0.42	1.0	0.88	0.80
rbtree	BL	0.18	0.33	0.42	1.0	0.88	0.76
rbtree	BR	0.18	0.32	0.41	1.0	0.88	0.76
AVLTree	ST	0.37	0.43	0.43	1.0	0.73	0.97
AVLTree	BR	0.37	0.43	0.43	1.0	0.73	0.97
FibHeap	ST	0.02	0.33	0.41	1.0	1.0	0.77
FibHeap	BR	0.02	0.33	0.40	1.0	1.0	0.84
HeapArr	ST	-	-	-	0.99	0.99	0.99
HeapArr	BR	-	-	-	0.99	0.99	0.99
TreeMap	ST	0.35	0.50	0.41	0.99	0.80	0.96
TreeMap	BR	0.35	0.50	0.41	0.99	0.80	0.96

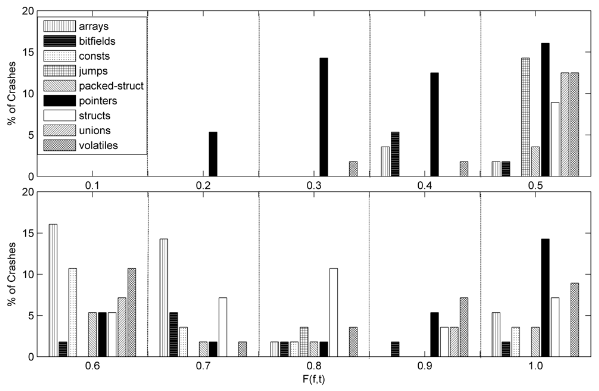

Fig. 2. $F(f,t)$ Scores for Some Compiler Features over Crashes

D. RQ4: Suppression Matters

Table V shows the strength of suppressors and triggers. The values under the column labeled "Min" show the single lowest $F(f,t)$ for each SUT and target type: the strength of the single most suppressing relationship (the strongest suppressor). The next columns show the *average minimum* $F(f,t)$ over, respectively, all features and all targets. The last three columns of the table, for comparison, show the same values for triggers, except that maximum $F(f,t)$ is used instead of minimum. In general, triggers are considerably stronger than suppressors, as we might expect: it is somewhat surprising that *any* faults or coverage targets lack at least one almost essential feature, particularly for API-based testing. The key observation, however, is that a large number of subjects not only had at least one suppressor strong enough to result in a large change in the difficulty of hitting a target, but that the *average* strongest suppressor for the average target over all subjects had $F < 0.45$. Recall that for an $F = 0.45$ suppressor, there will be 10% fewer tests hitting t in a population of tests that all contain f than in a population where f is present half the time. Given that many targets in automated testing have extremely low frequency (e.g., we found JavaScript failures with a rate as low as $\frac{1}{10^5}$ test cases, and file system testing at JPL found some rates lower than $\frac{1}{10^6}$ tests [6]), suppression almost certainly hides some faults.

Figure 2 shows the range of $F(f,t)$ for some features of C compilers. The graph is a histogram showing the percent of crashes for which each feature had $F(f,t)$ in a given 0.1-width range (shown by its max under the bars). While it is difficult to make out details at this scale, it is clear the crashes had a large range of $F(f,t)$ for many features: e.g., pointers and volatiles were very suppressing (0.2–0.3) for 15 and 2% of targets respectively, but critical triggers ($F \geq 0.9$) for another 15 and 8%.

E. Subject Details: Causes of Suppression

Suppression can be helpful for what might be termed dynamic program understanding — a cross between the general concept of "program understanding" which captures the notion of the "true" behavior of a system (as defined by the semantics of the program) and its observed behavior, which is determined partly by the true behavior and partly by the test suite used to observe the program in action. In practice, understanding a program's test suite can be nearly as important as understanding the program itself [19]. Knowing the suppressors for a single target can (akin to a fault localization approach like Tarantula [20]) help understand *that target*, whether it is a fault or simply a subtle piece of code. Understanding the causes of suppression for all targets can help in understanding why some test suites are more effective than others, or even provide insights into the nature of a software system. One problem with many dynamic understanding methods, e.g. invariants [21], is that it can be hard for users to decide if the results are "true" or an artifact of limited empirical data [22]. Suppression and triggering relationships do not require judgments as to whether they are universally valid, since their most obvious use is in tuning automated testing. We also expect that if used for oracle generation, our methods would pose easier problems than general invariants, in that they would always generate assertions of the form "t will never by observed in a test that has/does not have f."

1) YAFFS2: YAFFS2 serves as a useful introduction to common causes of suppression. Calling `close` is a moderately strong suppressor for almost 5% of mutants, and a marginal suppressor for another 15% of mutants. Interestingly, when `close` is a trigger it is either a very weak trigger or appears in at least 90% of all tests exposing a target. This tells us that closing files does not "help" with other behaviors. Closing files is useful in tests essentially only in order to test the code in the `yaffs_close` function. Examining the code for YAFFS2, we see that the function only calls functions that are called by numerous other functions. This explains why `close` is not very triggering, but the essence of its suppression is that `close`, like `pop` *reduces* the system state — it is a *consumer* or *simplifier* of state. There are fewer next behaviors of a system after a `close` than before. In the case of `close` or `pop` this results from the call being the "dual" of a producer (`open` or `push`). Similarly, both `fsync` and `sync` suppress about 7% of all mutants, and are in the top 6 most frequent suppressors because they empty out buffers.

A second interesting observation about YAFFS2 suppression is that it seems to show the effects of feedback [5], [4] in suppression. Random testing with feedback uses the current state of the system to select a next API call to make, avoiding invalid extensions of a test. Some API calls serve as "guards" for a large number of other calls: a single call to `open` enables a large number of other calls (e.g., `read`, `write`, `fstat`, `close`) that require a valid file handler. If a target does not require file operations — for example, numerous bugs involving directory manipulation — it will be "suppressed" by `open` because calling `open` can change the probability of calling `mkdir` (the trigger) from $\frac{1}{10}$ to $\frac{1}{20}$.

2) C Compilers: There was considerable diversity in the way that features of C programs interacted with compiler bugs (Figure 2). For example, GCC 4.0.0 has a crash bug ("internal compiler error: in c_common_type, at c-typeck.c:529") for which arrays are a trigger ($F = 0.64$) and pointers are a suppressor ($F = 0.27$). Another bug in the same compiler ("internal compiler error: in make_decl_rtl, at varasm.c:868") has both pointers and arrays as triggers with $F = 0.99$. Yet another bug in this compiler requires pointers ($F = 0.97$) but is indifferent to arrays. Other compilers in our test suite contain bugs that require arrays and that are suppressed by pointers. This is, in itself, quite interesting given that arrays in C are quite close to "syntactic sugar" for pointers.

Out of the 20 compiler bugs that had the strongest suppressors, pointers were the most common suppressor, occurring 13 times. We speculate that pointers suppress crashes by causing compiler optimization passes to operate in a more conservative fashion. Pointer suppression is different than the most common YAFFS2 suppression effects, in that it is difficult to claim that pointers make a C program's state simpler. The problem is that pointers simplify the behavior in some respects while complicating it in other respects (hence serving as a common triggers as well). Out of the remaining 7 suppressors, none occurred more than once.

The single strongest suppressor that we saw was the "argc" feature supported by Csmith, which suppressed an out-of-memory condition in Clang 2.7 with $F = 0.08$. Csmith's "argc" feature causes it to change the prototype of `main` so that the random program accepts command-line parameters, and it also adds about two lines of code that conditionally sets a flag telling the generated code to print some additional details about the computation it performs. (By default, a program generated by Csmith prints a single checksum.) Without the "argc" feature, the flag variable is unconditionally false. A smart compiler can recognize this and use it to perform more aggressive optimizations. However, Clang 2.7 contains a bug that causes its loop strength reduction pass to enter an infinite allocating loop, eventually running out of RAM. This bug is triggered by the fact that the generated code's flag variable is a compile-time constant value, and is suppressed when this flag's value is indeterminate at compile time due to command line processing. Many causes of suppression are this idiosyncratic, and thus almost impossible to predict without a deep understanding of the target behavior, which is almost certainly not available for unknown faults. Knowing such idiosyncratic relationships can directly assist random testing engineers in tuning a tester to target certain behaviors.

3) JavaScript Shells: Because it exhibited failures varying widely (by three orders of magnitude) in frequency, the `js` 1.6 shell demonstrates particularly well one source of suppression: fault masking. The single most frequent suppressor for 1.6 (suppressing 11 failures versus triggering 3) was assignment of an ID to an l-value, which was a moderate trigger for three very frequent failures, including the single most common failure (with over 4,000 instances). The F value for this feature as a *suppressor* decreased (recall that this means it was a stronger suppressor) as the rarity of failures increased, and it was a suppressor for two of the three rarest failures. Examining the code, there does not appear to be any link between the feature and the failures other than that the feature causes `js` to fail in one of the three "common" ways before the less frequent failures (all requiring longer executions on average) can take place. Complex systems will almost always have multiple faults that differ in frequency of appearance; in our experience, for compilers and JavaScript engines, failure rates for different faults typically exhibit a power law curve and sometimes a "double power law" curve [23]. Masking may therefore merit special attention as a source of suppression.

4) HTML Parser: None of the important features for HTML parsing were very surprising. That '>' was a suppressor for 34% of statements was interesting: some parse errors, e.g. nested unclosed parenthetical constructs, are hard to generate with a purely random mix.

5) SGLIB rbtree: Calls that add items to a container are, while redundant (and thus weaker triggers), needed in order to produce any interesting behavior, and suppress essentially no behaviors. While deletion calls are also interchangeable, however, they serve to suppress roughly as often as they trigger; care must be taken in automated container class testing to avoid over-inclusion of deletes.

V. Using Suppression Information to Predict, Produce, and Understand Behaviors

As an example of a practical application of suppression to dynamic program understanding, we consider the analysis of *never-paired* program statements: s_1 and s_2 are a never-pair if, over a test suite, no one test executes *both* s_1 and s_2. Never-paired statements can be thought of as a dual notion to the idea of *dynamic basic blocks*, statements that always have the same coverage value for each test [24]. Never-pairs are interesting in that they describe potential behavioral interactions that are never explored. (Never pairs are a particular example of an *extended invariant* [25].) If the behaviors are compatible but not explored, they may expose faults when combined; if the behaviors are not possible in the same test, for reasons other than trivial CFG relationships, the fact that they cannot be paired indicates partitioning of program behaviors. Unfortunately, the primary cause for never-pairing is the size of a test suite. If the hit rate for s_1 is r_1 and the hit rate for s_2 is r_2, for a suite of size k, if $r_1 \cdot r_2 \cdot k$ is low, we do not *expect* there to be any co-appearances for s_1 and s_2 even if they are compatible. If r_1 and r_2 are sufficiently low, running more tests may not be a practical solution to this problem: if both are "1 in a million" coverage targets, running 1 trillion tests may not be feasible. We would like to predict whether we can pair statements, in order to 1) identify statements that genuinely partition behavior, as important artifacts for program/suite understanding and 2) motivate directed searches for possible pairings, via symbolic/concolic or statistical techniques.

Suppression provides us such a prediction mechanism. By comparing suppressors and triggers for s_1 and s_2 that we would not expect to have co-detected in a suite, we can predict whether a larger suite will co-detect the features. For the `js` 1.8.5 coverage results, we examined a random sample of 300 pairs of never-paired statements over the first 5,000 tests executed, such that 1) s_1 and s_2 were both covered, 2) s_1 and s_2 had suppressors and/or triggers, 3) the rates did not predict any hits within these 5,000 tests, and 4) the rates *did* predict common hits for the full suite of $\sim$24,000 tests. In other words, we chose cases where 5,000 tests did not provide useful insight into whether the statements were truly difficult to pair, but where the remainder of the test suite was likely to resolve the question. To attempt to predict whether we would actually hit the statements together, we simply checked s_1 and s_2 for conflicting features — cases where f is a suppressor for s_1 and a trigger for s_2, or vice versa. If the statements had conflicting features, we predicted no hits for the remainder of the test suite. Even this naive approach (as opposed to using all suppressor and trigger information for s_1 and s_2 with a statistical model or machine learning algorithm to predict an expected hit rate) was correct over 66% of the time. Additionally, we were able to use the full suppression and trigger information for pairs to build "ideal" configurations for finding common occurrences, and prove some rarely appearing s_1, s_2 never-pairs spurious.

Moreover, examining the suppressors and triggers for our samples explained *why* some statements were difficult to execute together in the same test. For example, we found two statements handling destructuring assignments (one in `jsemit.cpp` and one in `jsparse.cpp`), both firing for unusual l-value choices that appear rarely in tests. Both statements "need" some form of destructuring assignment, but they require different arities in the assignment (as well as other complex conditions to prevent other code from handling the assignment). Because both statements are rarely executed in general, choosing *only* the arity appropriate is practically required to see the behavior — tests containing both arities are unlikely to manage to conjoin arity and conditions and hit either target. We realized this after seeing that the arity-one case is an $F = 0.30$ suppressor for one statement and a $F = 0.67$ trigger for the other.

VI. Suppression and Testing

This paper establishes that suppression is an important factor in random testing. What about other testing approaches?

A. Suppression and Other Test Methods

Suppression almost certainly matters for Bounded Exhaustive Testing (BET). While complete BET will, by definition, cover all test cases within a bound that omit some feature, suppression can make the *expected time* until a particular fault is detected much higher. In general, the expected time before detection of a fault in BET with tests of length k is controlled by the failure rate over pure random tests of length k [26].

For incomplete BET, or BET with abstraction, which is the only practical approach for realistic systems with very large branching factors (all of our non-container class examples), the effect of suppression has likely been seen in practice. In performing non-exhaustive model checking of a file system used in JPL's Curiosity rover, Groce noted that model checking where directory operations were disallowed could dramatically improve path coverage for file operations (and vice versa) [19]. The effectiveness of swarm verification [27] is partly based on reordering of transitions in model checking. For a depth-first search with limited memory/time, placing a transition last is very much like removing it from the feature set for a model checking run — which can reduce suppression.

As Jin and Orso have shown, symbolic testing can scale better for finding some faults when given a call sequence [28]. In their work, the aim is to reproduce a failure observed in the field, but the observation that KLEE [29] can reproduce failures much more easily given a call sequence than from point of failure information only suggests that one approach to scaling constraint-based approaches to complex systems such as YAFFS2 is to begin concolic testing from "skeletons" of potential test cases, rather than introducing a costly case split over various API calls at each step. Using suppression relationships should make it possible to avoid the combinatoric explosion produced by considering all feature combinations.

B. Mitigating Suppression

Swarm testing [1] works because it helps mitigate some effects of suppression by omitting features from tests. Swarm as practiced so far is a relatively simple strategy, however, and the purely random, uniform approach to configuration generation taken in swarm does not precisely fit the data. Swarm thus far assumes that 1) including and excluding features are equally useful (thus sets a 50% inclusion rate) and 2) all features should be treated equally. However some features are mostly triggers, some are mostly suppressors, and triggering is both much stronger and more frequent than suppression. Swarm also thus far fails to exploit causes of suppression (e.g., consumption, feedback, and fault masking).

The obvious response would be to use feedback, based on information such as that included in this paper. After an initial series of uniform 50% inclusion tests, fit the rate for each feature to its frequency and strength as a trigger or suppressor, etc. We speculate, however, that this might *decrease* effectiveness in practice. Tuning feature inclusion this way specializes the tester in *hitting targets that have already been hit*, while the desired goal is to hit targets that have not been hit. These targets may have been missed because they are masked by many already hit features, or because they depend on configurations not yet generated that are different from those best for hit targets. We therefore propose a mixed feedback strategy: on the one hand, re-hitting rarely hit targets may lead to coverage of further "nearby" targets. On the other hand, using "reverse" feedback to explore configurations "far" from the space in which easily hit targets reside may find new targets better. Initial experiments show that for reverse feedback, simply taking the most common failures and reducing the rates for triggers and increasing the rates for suppressors is not that helpful for `js` 1.6: some triggers for these faults are useful for getting *any* interesting behavior. Even with large amounts of data, it is hard to distinguish general triggers of useful behavior from triggers unique to only a subset of behaviors.

To reduce the "data gathering" phase of testing, we propose attempting to *statically* identify suppressors. Using typestate information, it should be possible to identify at least some producers and consumers, e.g. `close`. Similarly, calls that deallocate memory or reduce the value of indices into arrays (such as `sync`) can be detected. Cheap, imprecise analysis is acceptable as the properties are only used to tune feature selection. Over time, test results should identify incorrect assumptions — if a proposed "suppressor" suppresses few targets, its frequency can be increased.

VII. Related Work

In previous work on swarm testing [1] we suggested a formal notion of suppressors and proposed that suppression might explain the effectiveness of swarm. This paper extends those results by formalizing and evaluating the importance of (and causes of) suppression for a larger set of subject programs and targets, as well as showing a mismatch between swarm's configuration assumptions and actual ideal feature probabilities.

The investigations in this paper can also be seen as generalizing the ideas behind combinatorial testing [30]. Where Kuhn et al. consider the case when "behavior is not dependent on complex event sequences and variables have a small set of discrete values," we generalize to systems with more complex behaviors and a large set of inputs by replacing combinations of input parameters with more general constraints on test contents. Our work is also similar in spirit to (though more general than) key work in fault localization, in particular Tarantula [20], [31] and similar methods that use statistical relationships to find code relevant to a fault. Suppressors should have low "suspiciousness" scores by Tarantula-like methods. Applying Tarantula-like analysis to all targets, rather than just faults, to our knowledge, has not previously been considered.

VIII. Conclusions

In this paper, we used large-scale random testing to statistically investigate the relationships between test *features* (controllable aspects of a test) and *targets* (things we are interested in observing in a test). In particular, we looked at how features can *suppress* (make less likely) a target behavior. Investigating a variety of real-world programs we showed that:

- **RQ1:** Features tend to suppress at least some targets.
- **RQ2:** Large numbers of targets have suppressors.
- **RQ3:** The average number of suppressors (and triggers) for each target was strikingly and surprisingly similar across non-toy SUTs. The average was ≥ 1.0 in all cases, and had a standard deviation far smaller than that of the number of features.
- **RQ4:** Most subjects had at least one target with a very strong ($F < 0.20$) suppressor; average targets had suppressors able to cause a $> 10\%$ difference in hit rates.

Suppression therefore tends to cause some faults in systems to be difficult to detect — tests must not only include the right calls/grammar elements/etc., but they *must omit* other features to have a good chance of triggering many behaviors. Suppression likely applies not only to random testing but to bounded exhaustive testing and model checking. We suggest some methods for mitigating suppression, but believe that at present it poses an open problem for automated testing and results in undetected faults.

References

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

[2] N. Siegmund, S. S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *International Conference on Software Engineering*, 2012, pp. 166–177.

[3] 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.

[4] 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.

[5] 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.

[6] A. Groce, G. Holzmann, R. Joshi, and R.-G. Xu, "Putting flight software through the paces with testing, model checking, and constraint-solving," in *Workshop on Constraints in Formal Verification*, 2008, pp. 1–15.

[7] T. Ball, "A theory of predicate-complete test coverage and generation," in *Formal Methods for Components and Objects*, 2004, pp. 1–22.

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

[9] "Yaffs: A flash file system for embedded use," http://www.yaffs.net/.

[10] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*, 2005, pp. 402–411.

[11] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *International Conference on Software Engineering*, 2010, pp. 435–444.

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

[13] "The Lobo project," http://lobobrowser.org.

[14] M. Vittek, P. Borovansky, and P.-E. Moreau, "A simple generic library for C," in *Int. Conference on Software Reuse*, 2006, pp. 423–426.

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

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

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

[18] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

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

[20] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.

[21] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *International Conference on Software Engineering*, 1999, pp. 213–224.

[22] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding user understanding: determining correctness of generated program invariants," in *International Symposium on Software Testing and Analysis*, 2012, pp. 188–198.

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

[24] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *International Conference on Software Engineering*, 2006, pp. 82–91.

[25] M. A. Alipour and A. Groce, "Extended program invariants: applications in testing and fault localization," in *Workshop on Dynamic Analysis*, 2012, pp. 7–11.

[26] J. Andrews, Y. R. Zhang, and A. Groce, "Comparing automated unit testing strategies," Department of Computer Science, University of Western Ontario, Tech. Rep. 736, December 2010.

[27] G. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 845–857, 2011.

[28] W. Jin and A. Orso, "BugRedux: reproducing field failures for in-house debugging," in *International Conference on Software Engineering*, 2012, pp. 474–484.

[29] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating System Design and Implementation*, 2008, pp. 209–224.

[30] 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.

[31] D. L. Lucia, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *International Conference on Software Maintenance*, 2010, pp. 1–10.