# Taming Compiler Fuzzers

Yang Chen Alex Groce[†] Chaoqiang Zhang[†] Weng-Keen Wong[†]
Xiaoli Fern[†] Eric Eide John Regehr

University of Utah
Salt Lake City, UT

[†]Oregon State University
Corvallis, OR

chenyang@cs.utah.edu agroce@gmail.com zhangch@onid.orst.edu wong@eecs.oregonstate.edu
xfern@eecs.oregonstate.edu eeide@cs.utah.edu regehr@cs.utah.edu

## Abstract

Aggressive random testing tools ("fuzzers") are impressively effective at finding compiler bugs. For example, a single test-case generator has resulted in more than 1,700 bugs reported for a single JavaScript engine. However, fuzzers can be frustrating to use: they indiscriminately and repeatedly find bugs that may not be severe enough to fix right away. Currently, users filter out undesirable test cases using ad hoc methods such as disallowing problematic features in tests and grepping test results. This paper formulates and addresses the *fuzzer taming problem*: given a potentially large number of random test cases that trigger failures, order them such that diverse, interesting test cases are highly ranked. Our evaluation shows our ability to solve the fuzzer taming problem for 3,799 test cases triggering 46 bugs in a C compiler and 2,603 test cases triggering 28 bugs in a JavaScript engine.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.4 [*Programming Languages*]: Processors—compilers; H.3.3 [*Information Storage and Retrieval*]: Information Search and Retrieval—selection process

***Keywords*** compiler testing; compiler defect; automated testing; fuzz testing; random testing; bug reporting; test-case reduction

## 1. Introduction

Modern optimizing compilers and programming language runtimes are complex artifacts, and their developers can be under significant pressure to add features and improve performance. When difficult algorithms and data structures are tuned for speed, internal modularity suffers and invariants become extremely complex. These and other factors make it hard to avoid bugs. At the same time, compilers and runtimes end up as part of the trusted computing base for many systems. A code-generation error in a compiler for a critical embedded system, or an exploitable vulnerability in a widely deployed scripting language runtime, is a serious matter.

Random testing, or fuzzing, has emerged as an important tool for finding bugs in compilers and runtimes. For example, a single fuzzing tool, `jsfunfuzz` [31], is responsible for identifying more than 1,700 previously unknown bugs in SpiderMonkey, the JavaScript engine used in Firefox [32]. LangFuzz [14], a newer randomized testing tool, has led to the discovery of more than 500 previously unknown bugs in the same JavaScript engine. Google's proprietary ClusterFuzz effort "hammers away at it [Chrome, including its V8 JavaScript engine] to the tune of around fifty-million test cases a day," with apparent success [2]: "ClusterFuzz has detected 95 unique vulnerabilities since we brought it fully online at the end of last year [a four-month period]." For C compilers, Csmith [44] has identified more than 450 previously unknown bugs.

While fuzzers are powerful bug-finding tools, their use suffers from several drawbacks. The first problem is that failures due to random test cases can be difficult to debug. This has been largely solved by Delta Debugging [45], an automated greedy search for small failure-inducing test cases. In fact, there is some evidence that debugging based on short random test cases is easier than debugging based on human-created test cases [3]. A second problem is the sheer volume of output: an overnight run of a fuzzer may result in hundreds or thousands of failure-inducing test cases. Moreover, some bugs tend to be triggered much more often than others, creating needle-in-a-haystack problems. Figure 1 shows that some of the bugs studied in this paper were triggered thousands of times more frequently than others. Compiler engineers are an expensive and limited resource, and it can be hard for them to find time to sift through a large collection of highly redundant bug-triggering test cases. A third problem is that fuzzers are indiscriminate: they tend to keep finding more and more test cases that trigger noncritical bugs that may also already be known. Although it would be desirable to fix these bugs, the realities of software development—where resources are limited and deadlines may be inflexible—often cause low-priority bugs to linger unfixed for months or years. For example, in November 2012 we found 2,403 open bugs in GCC's bug database, considering priorities P1, P2, and P3, and considering only bugs of "normal" or higher severity. The median-aged bug in this list was well over two years old. If a fuzzer manages to trigger any appreciable fraction of these old, known bugs, its raw output will be very hard to use.

A typical workflow for using a random tester is to (1) start running the random tester against the latest version of the compiler; (2) go to bed; and (3) in the morning, sift through the new failure-inducing test cases, creating a bug report for each that is novel and important. Step 3 can be time-consuming and unrewarding. We know of several industrial compiler developers who stopped using Csmith not because it stopped finding bugs, but because step 3 became uneconomical. This paper represents our attempt to provide fuzzer users with a better value proposition.

Thus far, little research has addressed the problem of making fuzzer output more useful to developers. In a blog entry, Ruderman,



The definitive version was published in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, Jun. 2013, `http://doi.acm.org/10.1145/NNNNNNN.NNNNNNN`

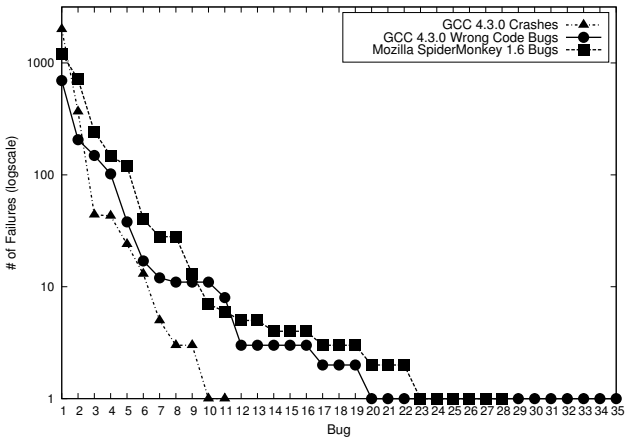


**Figure 1.** A fuzzer tends to hit some bugs thousands of times more frequently than others

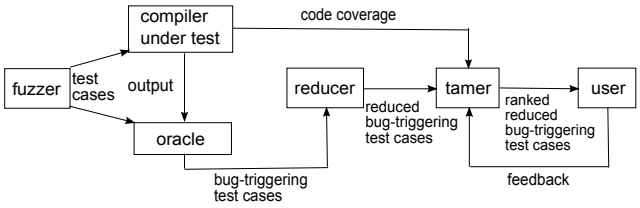


**Figure 2.** Workflow for a fuzzer tamer

the original author of the `jsfunfuzz` tool, reports using a variety of heuristics to avoid looking at test cases that trigger known bugs, such as turning off features in the test-case generator and using tools like `grep` to filter out test cases triggering bugs that have predictable symptoms [33]. During testing of mission file systems at NASA [12], Groce et al. used hand-tuned rules to avoid repeatedly finding the same bugs during random testing—e.g., "ignore any test case with a reset within five operations of a rename."

We claim that much more sophisticated automation is feasible and useful. In this paper we describe and evaluate a system that does this for two fuzzers: one for JavaScript engines, the other for C compilers. We characterize the *fuzzer taming problem*:

> Given a potentially large collection of test cases, each of which triggers a bug, rank them in such a way that test cases triggering distinct bugs are early in the list.
>
> **Sub-problem:** If there are test cases that trigger bugs previously flagged as undesirable, place them late in the list.

Ideally, for a collection of test cases that triggers $N$ distinct bugs (none of which have been flagged as undesirable), each of the first $N$ test cases in the list would trigger a different bug. In practice, perfection is unattainable because the problem is hard and also because there is some subjectivity in what constitutes "distinct bugs." Thus, our goal is simply to improve as much as possible upon the default situation where test cases are presented to users in effectively random order.

Figure 2 shows the workflow for a fuzzer tamer. The "oracle" in the figure detects buggy executions, for example by watching for crashes and by running the compiler's output against a reference compiler's output. Our rank-ordering approach was suggested by the prevalence of ranking approaches for presenting alarms produced by static analyses to users [18, 19].

Our contributions are as follows. First, we frame the fuzzer taming problem, which has not yet been addressed by the research community, as far as we are aware. Second, we make and exploit the observation that automatic triaging of test cases is strongly synergistic with automated test-case reduction. Third, based on the insight that bugs are highly diverse, we exploit diverse sources of information about bug-triggering test cases, including features of the test case itself, features from execution of the compiler on the test case, and features from the compiler's output. Fourth, we show that diverse test cases can be ranked highly by first placing test cases in a metric space and then using the *furthest point first* (FPF) technique from machine learning [10]. The more obvious approach to fuzzer taming is to use a clustering algorithm [27] to place tests into homogeneous groups, and then choose a representative test from each cluster. We show that FPF is both faster and more effective than clustering for all of our case studies. Finally, we show that our techniques can effectively solve the fuzzer taming problem for 2,603 test cases triggering 28 bugs in a JavaScript engine and 3,799 test cases triggering 46 bugs in a C compiler. Using our methods over this test suite, a developer who inspects the JavaScript engine's test cases in ranked order will more quickly find cases that trigger the 28 bugs found during the fuzzing run. In comparison to a developer who examines cases in a random order, the developer who inspects in ranked order will be 4.6× faster. For wrong-code bugs and crash bugs in the C compiler, the improvements are 2.6× and 32×, respectively. Even more importantly, users can find many more distinct bugs than would be found with a random ordering by examining only a few tens of test cases.

Taming a fuzzer differs from previous efforts in duplicate bug detection [37, 38, 42] because user-supplied metadata is not available: we must rely solely on information from failure-inducing test cases. Compared to previous work on dealing with software containing multiple bugs [15, 22, 28], our work differs in the methods used (ranking bugs as opposed to clustering), in the kinds of inputs to the machine learning algorithms (diverse, as opposed to just predicates or coverage information), and in its overall goal of taming a fuzzer.

## 2. Approach

This section describes our approach to taming compiler fuzzers and gives an overview of the tools implementing it.

### 2.1 Definitions

A *fault* or *bug* in a compiler is a flaw in its implementation. When the execution of a compiler is influenced by a fault—e.g., by wrong or missing code—the result may be an *error* that leads to a *failure* detected by a test oracle. In this paper, we are primarily concerned with two kinds of failures: (1) compilation or interpretation that fails to follow the semantics of the input source code, and (2) compiler crashes. The goal of a compiler fuzzer is to discover source programs—test cases—that lead to these failures. The goal of a *fuzzer tamer* is to rank failure-inducing test cases such that any prefix of the ranked list triggers as many different faults as possible. Faults are not directly observable, but a fuzzer tamer can estimate which test cases are related by a common fault by making an assumption: the more "similar" two test cases, or two executions of the compiler on those test cases, the more likely they are to stem from the same fault [23].

A *distance function* maps any pair of test cases to a real number that serves as a measure of similarity. This is useful because our goal is to present fuzzer users with a collection of highly dissimilar test cases. Because there are many ways in which two test cases can be similar to each other—e.g., they can be textually similar, cause similar failure output, or lead to similar executions of the compiler—our work is based on several distance functions.

### 2.2 Ranking Test Cases

Our approach to solving the fuzzer taming problem is based on the following idea.

> **Hypothesis 1:** If we (1) define a distance function between test cases that appropriately captures their static and dynamic characteristics and then (2) sort the list of test cases in *furthest point first* (FPF) order, then the resulting list will constitute a usefully approximate solution to the fuzzer taming problem.

If this hypothesis holds, the fuzzer taming problem is reduced to defining an appropriate distance function. The FPF ordering is one where each point in the list is the one that maximizes the distance to the nearest of all previously listed elements; it can be computed using a greedy algorithm [10]. We use FPF to ensure that diverse test cases appear early in the list. Conversely, collections of highly similar test cases will be found towards the end of the list.

Our approach to ignoring known bugs is based on the premise that fuzzer users will have labeled some test cases as exemplifying these bugs; this corresponds to the "feedback" edge in Figure 2.

> **Hypothesis 2:** We can lower the rank of test cases corresponding to bugs that are known to be uninteresting by "seeding" the FPF computation with the set of test cases that are labeled as uninteresting.

Thus, the most highly ranked test case will be the one maximizing its minimum distance from any labeled test case.

### 2.3 Distance Functions for Test Cases

The fundamental problem in defining a distance function that will produce good fuzzer taming results is that we do not know what the trigger for a generic compiler bug looks like. For example, one C compiler bug might be triggered by a struct with a certain sequence of bitfields; another bug might be triggered by a large number of local variables, which causes the register allocator to spill. Our solution to this fundamental ambiguity has been to define a variety of distance functions, each of which we believe will usefully capture some kinds of bug triggers. This section describes these distance functions.

***Levenshtein Distance*** Also known as edit distance, the Levenshtein distance [20] between two strings is the smallest number of character additions, deletions, and replacements that suffices to turn one string into the other. For every pair of test cases we compute the Levenshtein distance between the following, all of which can be treated as plain text strings:

- the test cases themselves;
- the output of the compiler as it crashes (if any); and
- the output of Valgrind [25] on a failing execution (if any).

Computing Levenshtein distance requires time proportional to the product of the string lengths, but the constant factor is small (a few tens of instructions), so it is reasonably efficient in practice.

***Euclidean Distance*** Many aspects of failure-inducing test cases, and of executions of compilers on these test cases, lend themselves to summarization in the form of *feature vectors*. For example, consider this reduced JavaScript test case, which triggers a bug in SpiderMonkey 1.6:

```
__proto__=__parent__
new Error(this)
```

Lexing this code gives eight tokens, and a feature vector based on these tokens contains eight nonzero elements. The overall vector contains one element for every token that occurs in at least one test case, but which does not occur in every test case, out of a batch of test cases that is being processed by the fuzzer tamer. The elements in the vector are based on the number of appearances of each token in the test case. We construct lexical feature vectors for both C and JavaScript.

Given two $n$-element vectors $v_1$ and $v_2$, the Euclidean distance between them is:

$$\sqrt{\sum_{i=1..n} (v_1[i] - v_2[i])^2}$$

For C code, our intuition was that lexical analysis in some sense produced shallower results than it did for JavaScript. To compensate, we wrote a Clang-based detector for 45 additional features that we guessed might be associated with compiler bugs. These features include:

- common types, statement classes, and operator kinds;
- features specific to aggregate data types such as structs with bitfields and packed structs;
- obvious divide-by-zero operations; and
- some kinds of infinite loops that can be detected statically.

In addition to constructing vectors from test cases, we also constructed feature vectors from compiler executions. For example, the *function coverage* of a compiler is a list of the functions that it executes while compiling a test case. The overall feature vector for function coverage contains an element for every function executed while compiling at least one test case, but that is not executed while compiling all test cases. As with token-based vectors, the vector elements are based on how many times each function executed. We created vectors of:

- functions covered;
- lines covered;
- tokens in the compiler's output as it crashes (if any); and
- tokens in output from Valgrind (if any).

In the latter two cases, we use the same tokenization as with test cases (treating output from the execution as a text document), except that in the case of Valgrind we abstract some non-null memory addresses to a generic ADDRESS token. The overall hypothesis is that most bugs will exhibit some kind of dynamic signature that will reveal itself in one or more kinds of feature vector.

***Normalization*** Information retrieval tasks can often benefit from *normalization*, which serves to decrease the importance of terms that occur very commonly, and hence convey little information. Before computing distances over feature vectors, we normalized the value of each vector element using *tf-idf* [34]; this is a common practice in text clustering and classification. Given a count of a feature (token) in a test case or its execution (the "document"), the tf-idf is the product of the term-frequency (tf) and the inverse-document-frequency (idf) for the token. Term-frequency is the ratio of the count of the token in the document to the total number of tokens in the document. (For coverage we use number of times the entity is executed.) Inverse-document-frequency is the logarithm of the ratio of the total number of documents and the total number of documents containing the token: this results in a uniformly zero value for tokens appearing in all documents, which are therefore not included in the vector. We normalize Levenshtein distances by the length of the larger of the two strings, which helps handle varying sizes for test cases or outputs.

## 3. A Foundation for Experiments

To evaluate our work, we need a large collection of reduced versions of randomly generated test cases that trigger compiler bugs. Moreover, we require access to ground truth: the actual bug triggered by each test case. This section describes our approach to meeting these prerequisites.

### 3.1 Compilers Tested

We chose to test GCC 4.3.0 and SpiderMonkey 1.6, both running on Linux on x86-64. SpiderMonkey, best known as the JavaScript engine embedded in Firefox, is a descendant of the original JavaScript implementation; it contains an interpreter and several JIT compilers. Our selection of these particular versions was based on several considerations. First, the version that we fuzzed had to be buggy enough that we could generate useful statistics. Second, it was important that most of the bugs revealed by our fuzzer had been fixed by developers. This would not be the case for very recent compiler versions. Also, it turned out not to be the case for GCC 4.0.0, which we initially started using and had to abandon, since maintenance of its release branch—the 4.0.x series—terminated in 2007 with too many unfixed bugs.

### 3.2 Test Cases for C

We used the default configuration of Csmith [44] version 2.1.0, which over a period of a few days generated 2,501 test cases that crash GCC and 1,298 that trigger wrong-code bugs. The default configuration of Csmith uses swarm testing [13], which varies test features to improve fault detection and code coverage. Each program emitted by Csmith was compiled at `-O0`, `-O1`, `-O2`, `-Os`, and `-O3`.

To detect crash bugs, we inspected the return code of the main compiler process; any nonzero value was considered to indicate a crash. To detect wrong-code bugs, we employed *differential testing*: we waited for the compiler's output to produce a result different from the result of executing the output of a reference compiler. Since no perfect reference compiler exists, we approximated one by running GCC 4.6.0 and Clang 3.1 at their lowest optimization levels and ensuring that both compilers produced executables that, when run, had the same output. (We observed no mismatches during our tests.)

Csmith's outputs tend to be large, often exceeding 100 KB. We reduced each failure-inducing test case using C-Reduce [29], a tool that uses a generalized version of Delta debugging to heuristically reduce C programs. After reduction, some previously different tests became textually equivalent; this happens because C-Reduce tries quite hard to reduce identifiers, constants, data types, and other constructs to canonical values. For crash bugs, reduction produced 1,797 duplicates, leaving only 704 different test cases. Reduction was less effective at canonicalizing wrong-code test cases, with only 23 duplicate tests removed, leaving 1,275 tests to examine. In both cases, the typical test case was reduced in size by two to three orders of magnitude, to an average size of 128 bytes for crash bugs and 243 bytes for wrong-code bugs.

### 3.3 Test Cases for JavaScript

We started with the last public release of `jsfunfuzz` [31], a tool that, over its lifetime, has led to the discovery of more than 1,700 faults in SpiderMonkey. We modified `jsfunfuzz` to support swarm testing and then ran it for several days, accumulating 2,603 failing test cases. Differential testing of JavaScript compilers is problematic due to their diverging implementations of many of the most bug-prone features of JavaScript. However, `jsfunfuzz` comes with a set of built-in test oracles, including semantic checks (e.g., ensuring that compiling then decompiling code is an identity function) and watchdog timers to ensure that infinite loops can only result from faults. For an ahead-of-time compiler like GCC, it is natural to divide bugs into those that manifest at compile time (crashes) and those that manifest at run time (wrong-code bugs). This distinction makes less sense for a just-in-time compiler such as SpiderMonkey; we did not attempt to make it, but rather lumped all bugs into a single category. Test cases produced by `jsfunfuzz` were also large, over 100 KB on average. We reduced test cases using a custom reducer similar in spirit to C-Reduce, tuned for JavaScript. Reduction resulted in 854 duplicate test cases that we removed, leaving 1,749 test cases for input to the fuzzer taming tools. The typical failure-inducing test case for SpiderMonkey was reduced in size by more than three orders of magnitude, to an average size of 68 bytes.

### 3.4 Establishing Ground Truth

Perhaps the most onerous part of our work involved determining ground truth: the actual bug triggered by each test case. Doing this the hard way—examining the execution of the compiler for each of thousands of failure-inducing test cases—is obviously infeasible.

Instead, our goal was to create, for each of the 74 total bugs that our fuzzing efforts revealed, a patched compiler fixing only that bug. At that point, ground-truth determination can be automated: for each failure-inducing test case, run it through every patched version of the compiler and see which one changes its behavior. We only partially accomplished our goal. For a collection of arbitrary bugs in a large application that is being actively developed, it turns out to be very hard to find a patch fixing each bug, and only that bug.

For each bug, we started by performing an automated forward search to find the patch that fixed the bug. In some cases this patch (1) was small; (2) clearly fixed the bug triggered by the test case, as opposed to masking it by suppressing execution of the buggy code; and (3) could be back-ported to the version of the compiler that we tested. In other cases, some or all of these conditions failed to hold. For example, some compiler patches were extraordinarily complex, changing tens of thousands of lines of code. Moreover, these patches were written for compiler versions that had evolved considerably since the GCC 4.3.0 and SpiderMonkey 1.6 versions that are the basis for our experiments.

Although we spent significant effort trying to create a minimal patch fixing each compiler bug triggered by our fuzzing effort, this was not always feasible. Our backup strategy for assessing ground truth was first to approximately classify each test case based on the revision of the compiler that fixed the bug that it triggered, and second to manually inspect each test case in order to determine a final classification for which bug it triggered, based on our understanding of the set of compiler bugs.

### 3.5 Bug Slippage

When the original and reduced versions of a test case trigger different bugs, we say that *bug slippage* has occurred. Slippage is not hard to avoid for bugs that have an unambiguous symptom (e.g., "assertion violation at line 512") but it can be difficult to avoid for silent bugs such as those that cause a compiler to emit incorrect code. Although slippage is normally difficult to recognize or quantify, these tasks are easy when ground truth is available, as it is here.

Of our 2,501 unreduced test cases that caused GCC 4.3.0 to crash, almost all triggered the same (single) bug that was triggered by the test case's reduced version. Thirteen of the unreduced test cases triggered two different bugs, and in all of these cases the reduced version triggered one of the two. Finally, we saw a single instance of actual slippage where the original test case triggered one bug in GCC leading to a segmentation fault and the reduced version triggered a different bug, also leading to a segmentation fault. For the 1,298 test cases triggering wrong-code bugs in GCC, slippage during reduction occurred fifteen times.

For JavaScript, bug slippage was a more serious problem: 23% of reduced JavaScript test cases triggered a different bug than the original test case. This problem was not mitigated (as we had

originally hoped) by re-reducing test cases using the slower "debug" version of SpiderMonkey.

In short, bug slippage was a problem for SpiderMonkey 1.6 but not for GCC 4.3.0. Although the dynamics of test-case reduction are complex, we have a hypothesis about why this might have been the case. Test-case reduction is a heuristic search that explores one particular path through the space of all possible programs. This path stays in the subset of programs that trigger a bug and also follows a gradient leading towards smaller test cases. Sometimes, the trajectory will pass through the space of programs triggering some completely different bug, causing the reduction to be "hijacked" by the second bug. We would expect this to happen more often for a compiler that is buggier. Our observation is that GCC 4.3.0 is basically a solid and mature implementation whereas SpiderMonkey 1.6 is not—it contains many bugs in fairly basic language features.

## 4. Results and Discussion

For 1,979 reduced test cases triggering 46 bugs in GCC 4.3.0 and 1,749 reduced test cases triggering 28 bugs in SpiderMonkey 1.6, our goal is to rank them for presentation to developers such that diverse faults are triggered by test cases early in the list.

### 4.1 Evaluating Effectiveness using Bug Discovery Curves

Figures 3–8 present the primary results of our work using *bug discovery curves*. A discovery curve shows how quickly a ranking of items allows a human examining the items one by one to view at least one representative of each different category of items [26, 40]. Thus, a curve that climbs rapidly is better than a curve that climbs more slowly. Here, the items are test cases and categories are the underlying compiler faults. The top of each graph represents the point at which all faults have been presented. As shown by the y-axes of the figures, there are 28 SpiderMonkey bugs, 11 GCC crash bugs, and 35 GCC wrong-code bugs in our study.

Each of Figures 3–8 includes a baseline: the expected bug discovery curve without any fuzzer taming. We computed it by looking at test cases in random order, averaged over 10,000 orders. We also show the theoretical best curve where for $N$ faults each of the first $N$ tests reveals a new fault.

In each graph, we show in solid black the first method to find all bugs (which, in all of our examples, is also the method with the best area under the full curve). For GCC crash bugs and SpiderMonkey, this method also has the best climb for the first 50 tests, and for GCC wrong-code bugs, it is almost the best for the first 50 tests (and, in fact, discovers one more bug than the curve with the best area). For this best curve, we also show points sized by the log of the frequency of the fault; our methods do not always find the most commonly triggered faults first. Finally, each graph additionally shows the best result that we could obtain by ranking test cases using clustering instead of FPF, using X-means to generate clusterings by various features, sorting all clusterings by isolation and compactness, and using the centermost test for each cluster. (See Section 4.6 for details.)

### 4.2 Are These Results Any Good?

Our efforts to tame fuzzers would have clearly failed had we been unable to significantly improve on the baseline. On the other hand, there is plenty of room for improvement: our bug discovery curves do not track the "theoretical best" lines in Figures 3 and 7 for very long. For GCC crash bugs, however, our results are almost perfect.

Perhaps the best way to interpret our results is in terms of the value proposition they create for compiler developers. For example, if a SpiderMonkey team member examines 15 randomly chosen reduced test cases, he or she can expect them to trigger five different bugs. In contrast, if the developer examines the first 15 of our ranked tests, he or she will see 11 distinct bugs: a noticeable improvement.

### 4.3 Selecting a Distance Function

In Section 2 we described a number of ways to compute distances between test cases. Since we did not know which of these would work, we tried all of them individually and together, with Figures 3–8 showing our best results. Since we did not consider enough case studies to be able to reach a strong conclusion such as "fuzzer taming should always use Levenshtein distance on test case text and compiler output," this section analyzes the detailed results from our different distance functions, in hopes of reaching some tentative conclusions about which functions are and are not useful.

***SpiderMonkey and GCC Crash Bugs*** For these faults, the best distance function to use as the basis for FPF, based on our case studies, is the normalized Levenshtein distance between test cases plus normalized Levenshtein distance between failure outputs. Our tentative recommendation for bugs that (1) reduce very well and (2) have compiler-failure outputs is: use normalized Levenshtein distance over test-case text plus compiler-output text, and do not bother with Valgrind output or coverage information.

Given that using Levenshtein distance on the test-case text plus compiler output worked so well for both of these bug sets, where all faults had meaningful failure symptoms, we might expect using output or test-case text alone to also perform acceptably. In fact, Levenshtein distance based on test-case text alone (not normalized) performed moderately well for SpiderMonkey, but otherwise the results for these distance functions were uniformly mediocre at best. For GCC, using compiler output plus C features (Section 2.3) performed nearly as well as the best distance function, suggesting that the essential requirement is compiler output combined with a good representation of the test case, which may not be satisfied by a simple vectorization: vectorizing test case plus output performed badly for both GCC and SpiderMonkey.

Coverage-based methods worked fairly well for SpiderMonkey, appearing in six of the top ten functions and only two of the worst ten. Interestingly, these best coverage methods for SpiderMonkey all included both line and function coverage. Both coverage-based functions were uniformly mediocre for GCC crashes (coverage did not appear in any of the best ten or worst ten methods). For GCC, Valgrind was of little value, as most failures did not produce any Valgrind output. Memory-safety errors were more common in SpiderMonkey, so most test cases did produce Valgrind output; however, for the most part, adding the information to a distance function still made the function perform worse in the long run. Valgrind output alone performed extremely poorly in the long run for both GCC crashes and SpiderMonkey bugs.

***GCC Crash Bugs*** For these bugs, every distance function increased the area under the curve for examining less than 50 tests by a factor or four or better, compared to the baseline. Clearly there is a significant amount of redundancy in the information provided by different functions. All but five of the 63 distance functions we used were able to discover all bugs within at most 90 tests: a dramatic improvement over the baseline's 491 tests. Only Valgrind output alone performed worse than the baseline. The other four poorly performing methods all involved using vectorization of the test case, with no additional information beyond Valgrind output and/or test-case output.

GCC crash bugs were, however, our easiest target: there are only 11 crash outputs and 11 faults. Even so, the problem is not trivial, as the faults and outputs do not correspond perfectly—two faults have two different outputs, and there are two outputs that are each produced by two different faults. Failure output alone provides a great deal of information about a majority of the faults, and test-case distance completes the story.

***SpiderMonkey Bugs*** The story was less simple for SpiderMonkey bugs, where many methods performed poorly and seven methods per-

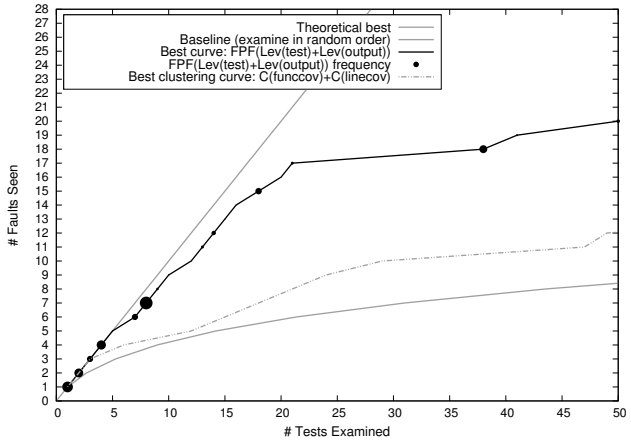


**Figure 3.** SpiderMonkey 1.6 bug discovery curves, first 50 tests

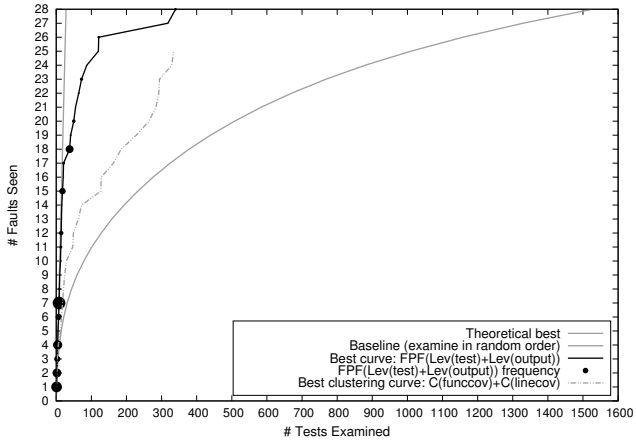


**Figure 4.** SpiderMonkey 1.6 bug discovery curves, all tests

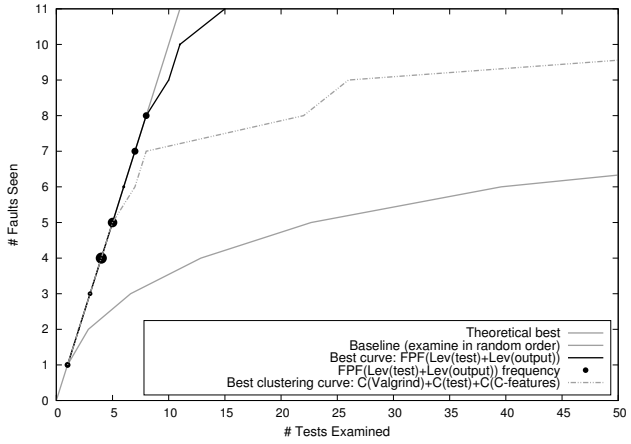


**Figure 5.** GCC 4.3.0 crash bug discovery curves, first 50 tests

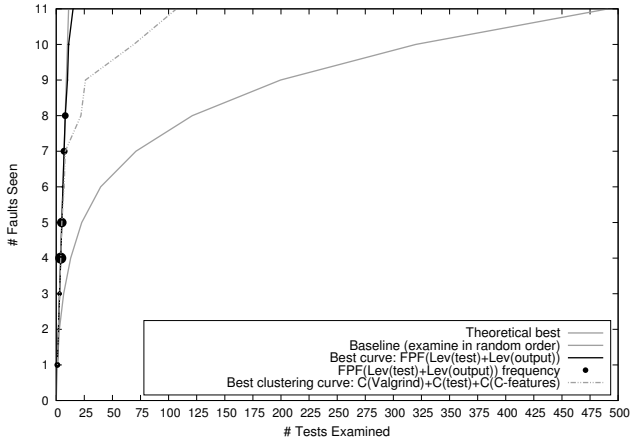


**Figure 6.** GCC 4.3.0 crash bug discovery curves, all tests

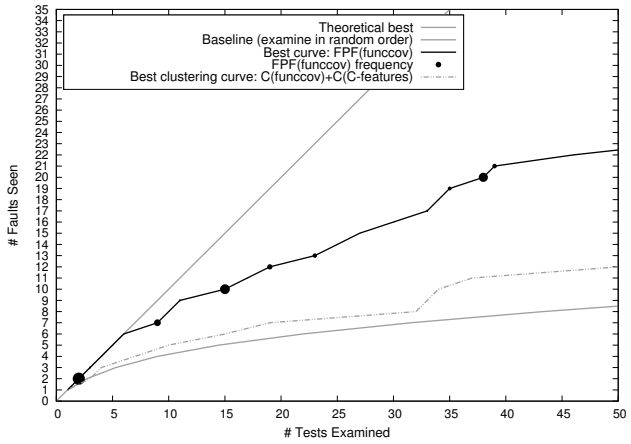


**Figure 7.** GCC 4.3.0 wrong-code bug discovery curves, first 50 tests

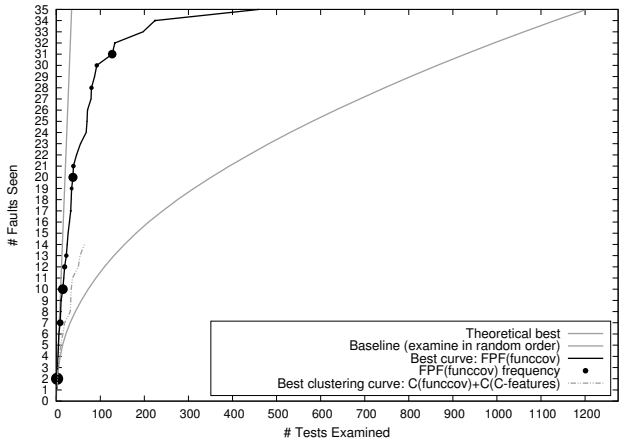


**Figure 8.** GCC 4.3.0 wrong-code bug discovery curves, all tests

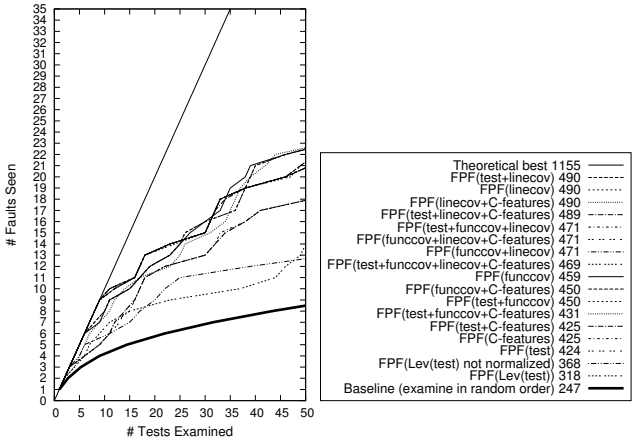


**Figure 9.** All bug discovery curves for GCC 4.3.0 wrong-code bugs, sorted by increasing area under the curve if examining only the 50 top-ranked test cases

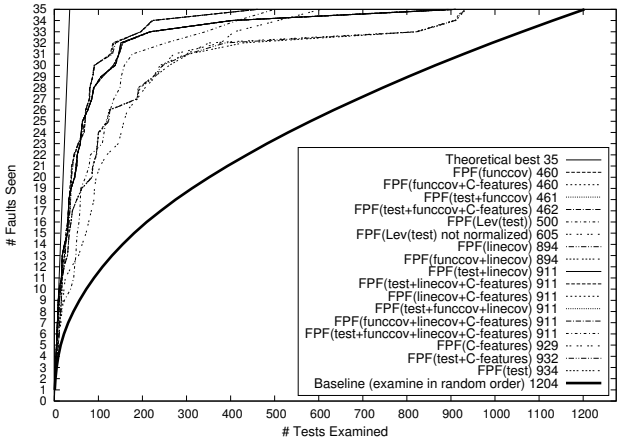


**Figure 10.** All bug discovery curves for GCC 4.3.0 wrong-code bugs, sorted by increasing number of test cases that must be examined to discover all faults

formed worse than the baseline. In this case, compiler output alone did not provide as much direct guidance: there were 300 different failure outputs, and only four of 28 faults had a unique identifying output. As a result, while compiler output alone performed very well over the first 50 tests (where it was one of the best five functions), it proved one of the worst functions for finding all faults, detecting no new faults between the 50th and 100th tests ranked. Test-case text by itself performed well for SpiderMonkey with Levenshtein distance, or when combined with line coverage, but performed badly as a vectorization without line coverage, appearing in six of the worst ten functions. As with GCC crashes, Valgrind output alone performed very badly, requiring a user to examine 1,506 tests to discover all bugs. Levenshtein-based approaches (whether over test case, compiler output, Valgrind output, or a combination thereof) performed very well over the first 50 tests examined.

***GCC Wrong-Code Bugs*** Wrong-code bugs in GCC were the trickiest bugs that we faced: their execution does not provide failure output and, in the expected case where the bug is in a "middle end" optimizer, the distance between execution of the fault and actual emission of code (and thus exposure of failure) can be quite long.

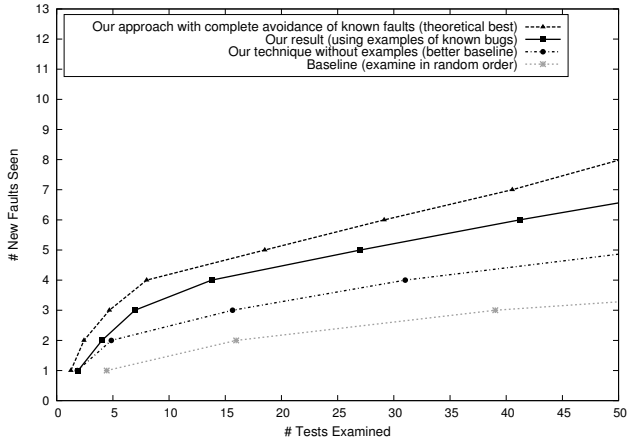


**Figure 11.** Avoiding known bugs in SpiderMonkey 1.6

For these bugs, the best method to use for fuzzer taming was less clear. Figures 9 and 10 show the performance of all methods that we tried, including a table of results sorted by area under the curve up to 50 tests (Figure 9) and number of test cases to discover all faults (Figure 10). It is clear that code coverage (line or function) is much more valuable here than with crash bugs, though Levenshtein distance based on test case alone performs well in the long run (but badly initially). Line coverage is useful for early climb, but eventually function coverage is most useful for discovering all bugs. Perhaps most importantly, given the difficulty of handling GCC wrong-code bugs, all of our methods perform better than the baseline in terms of ability to find all bugs, and provide a clear advantage over the first 50 test cases. We do not wish to overgeneralize from a few case studies, but these results provide hope that for difficult bugs, if good reduction is possible, the exact choice of distance function used in FPF may not be critical.

We were disappointed to see that Figures 9 and 10 show no evidence that our domain-specific feature detector for C programs is useful for wrong-code bugs; in the tables it appears as "C-Feature."

### 4.4 Avoiding Known Faults

In Section 2.2 we hypothesized that FPF could be used to avoid reports about a set of known bugs; this is accomplished by lowering the rankings of test cases that appear to be caused by those bugs. Figure 11 shows, for our SpiderMonkey test cases, an averaged bug discovery curve for the case where half of the bugs were assumed to be already known, and five test cases (or fewer, if five were not available) triggering each of those bugs were used to seed FPF. This experiment models the situation where, in the days or weeks preceding the current fuzzing run, the user has flagged these test cases and does not want to see more test cases triggering the same bugs. The curve is the average of 100 discovery curves, each corresponding to a different randomly chosen set of known bugs.

The topmost bug discovery curve in Figure 11 represents an idealized best case where all test cases corresponding to known bugs are removed from the set of test cases to be ranked. The second curve from the top is our result. The third curve from the top is also the average of 100 discovery curves; each corresponds to the case where the five (or fewer) test cases for each known bug are discarded instead of being used to seed the FPF algorithm, and then the FPF algorithm proceeds normally. This serves as a baseline: our result would have to be considered poor if it could not improve on this. Finally, the bottom curve is the "basic baseline" where the labeled test cases are again discarded, but then the remaining test cases are examined in random order.

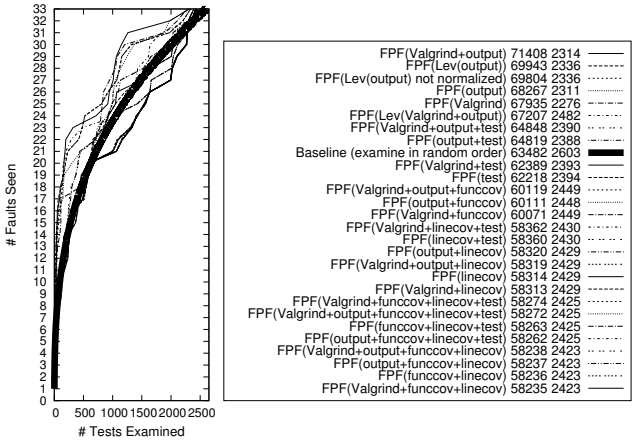


**Figure 12.** All bug discovery curves for SpiderMonkey 1.6 using unreduced test cases. It is difficult to significantly improve on the baseline without test case reduction.

As can be seen, our current performance for SpiderMonkey is reasonably good. Analogous results for GCC bugs (not included for reasons of space) were similar, but not quite as good for wrong-code bugs. We speculate that classification, rather than clustering or ranking, might be a better machine-learning approach for this problem if better results are required.

### 4.5 The Importance of Test-Case Reduction

Randomly generated test cases are more effective at finding bugs when they are large [1]. There are several reasons for this. First, large tests are more likely to bump into implementation limits and software aging effects in the system under test. Second, large tests amortize start-up costs. Third, undesirable feature interactions in the system under test are more likely to occur when a test case triggers more behaviors.

The obvious drawback of large random test cases is that they contain much content that is probably unrelated to the bug. They consequently induce long executions that are difficult to debug. Several random testers that have been used in practice, including McKeeman's C compiler fuzzer [24] and QuickCheck [6], have included built-in support for greedily reducing the size of failure-inducing inputs. Zeller and Hildebrandt [45] generalized and formalized this kind of test-case reduction as Delta Debugging.

While previous work has assumed that the consumer for reduced test cases is a human, our observation is that machine-learning-based methods can greatly benefit from reduced test cases. First, machine-learning algorithms can be overwhelmed by noisy inputs: reduced test cases have a vastly improved "signal to noise ratio." Second, a suitably designed test-case reducer has a canonicalizing effect.

Figure 12 shows the discovery curves for the FPF ordering on unreduced JavaScript test cases, accompanied by a table sorted by area under the discovery curve for all 2,603 unreduced tests (first column after distance function name) and also the number of tests required to discover all faults (second column). Note that this is a different baseline than in previous graphs, as there are no duplicates among the unreduced test cases. While all methods improve (slightly) on the baseline for finding all faults, it is difficult to consider these approaches acceptable: many methods produce an overall discovery curve that is worse than the baseline. Without test-case reduction, it is essentially impossible to efficiently find the more obscure SpiderMonkey bugs.

Based on these poor results, and on the fact that ranking unreduced test cases (which have much longer feature vectors) is expensive, our view is that attempting to tame a fuzzer without the aid of a solid test-case reducer is inadvisable. The most informative sources of information about root causes are rendered useless by overwhelming noise. Although we did not create results for unreduced GCC test cases that are analogous to those shown in Figure 12 (the line coverage vectors were gigantic and caused problems by filling up disks), we have no reason to believe the results would have been any better than they were for JavaScript.

### 4.6 Clustering as an Alternative to Furthest Point First

The problem of ranking test cases is not, essentially, a clustering problem. On the other hand, if our goal were simply to find a single test case triggering each fault, an obvious approach would be to cluster the test cases and then select a single test from each cluster, as in previous approaches to the problem [9, 28]. The FPF algorithm we use is itself based on the idea of approximating optimal clusters [10]; we simply ignore the clustering aspect and use only the ranking information.

Our initial approach to taming compiler fuzzers was to start with the feature vectors described in Section 2.3 and then, instead of ranking test cases using FPF, use X-means [27] to cluster test cases. A set of clusters does not itself provide a user with a set of representative test cases, however, nor a ranking (since not all clusters are considered equally likely to represent true categories). Our approach therefore followed clustering by selecting the member of each cluster closest to its center as that cluster's representative test. We ranked each test by the quality of its cluster, as measured by compactness (whether the distance between tests in the cluster is small) and isolation (whether the distance to tests outside the cluster is large) [40]. This approach appeared to be promising as it improved considerably on the baseline bug discovery curves (Figures 3–8).

We next investigated the possibility of independently clustering different feature vectors, then merging the representatives from these clusterings [36], and ranking highest those representatives appearing in clusterings based on multiple feature sets. This produced better results than our single-vector method, and it was also more efficient, as it did not require the use of large vectors combining multiple features. This approach is essentially a completely unsupervised variation (with the addition of some recent advances in clustering) of earlier approaches to clustering test cases that trigger the same bug [9]. Our approach is unsupervised because we exploit test-case reduction as a way to select relevant features, rather than relying on the previous approaches' assumption that features useful in predicting failure or success would also distinguish failures from each other.

However, in comparison to FPF for all three of our case studies, clustering was (1) significantly more complex to use, (2) more computationally expensive, and (3) most importantly, less effective. The additional complexity of clustering should be clear from our description of the algorithm, which omits details such as how we compute normalized isolation and compactness, the algorithm for merging multiple views, and (especially) the wide range of parameters that can be supplied to the underlying X-means algorithm.

Table 1 compares runtimes, with the time for FPF including the full end-to-end effort of producing a ranking and the clustering column only showing the time for computing clusters using X-means, with settings that are a compromise between speed and effectiveness. (Increased computation time to produce "more accurate" clusters in our experience had diminishing returns after this point, which allowed up to 40 splits and a maximum of 300 clusters.) Computing isolation and compactness of clusters and merging clusters to produce a ranking based on multiple feature vectors adds additional significant overhead to the X-means time shown, if multiple clusterings are combined, but we have not measured this time because our implementation is highly unoptimized Python (while X-means

| Program / Feature | Time (s) FPF | Clustering | Figures |
|---|---|---|---|
| SpiderMonkey / Valgrind | 8.27 | 23.68 | – |
| SpiderMonkey / output | 8.38 | 46.71 | – |
| SpiderMonkey / test | 8.12 | 94.26 | – |
| SpiderMonkey / funccov | 9.56 | 227.78 | 3, 4 |
| SpiderMonkey / linecov | 48.29 | 1,594.04 | 3, 4 |
| SpiderMonkey / Lev. test+output | 998.21 | N/A | 3, 4 |
| GCC crash bugs / output | 0.08 | 0.71 | – |
| GCC crash bugs / Valgrind | 0.09 | 0.75 | 5, 6 |
| GCC crash bugs / C-Feature | 0.10 | 1.95 | 5, 6 |
| GCC crash bugs / test | 0.14 | 15.12 | 5, 6 |
| GCC crash bugs / funccov | 1.37 | 162.22 | – |
| GCC crash bugs / linecov | 18.70 | 2,021.08 | – |
| GCC crash bugs / Lev. test+output | 75.07 | N/A | 5, 6 |
| GCC wrong-code bugs / C-Feature | 0.49 | 4.26 | 7, 8 |
| GCC wrong-code bugs / test | 0.72 | 67.72 | – |
| GCC wrong-code bugs / funccov | 4.12 | 1,046.07 | 7, 8 |
| GCC wrong-code bugs / linecov | 60.60 | 7,127.42 | – |
| GCC wrong-code bugs / Lev. test | 667.21 | N/A | – |

**Table 1.** Runtimes for FPF versus clustering

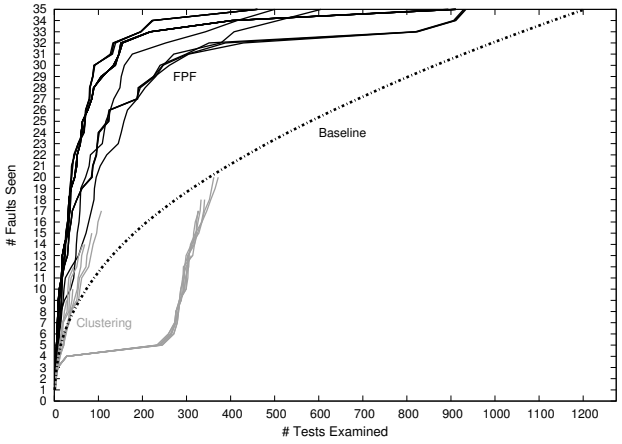


**Figure 13.** GCC 4.3.0 wrong-code bug clustering comparison

is a widely used tool written in C). Because the isolation and compactness computations require many pairwise distance results, an efficient implementation should be approximately equal in time to running FPF. The final column of the table lists the figures in this paper that show a curve based on the indicated results. If a curve relies on multiple clusterings, its generation time is (at least) the sum of the clustering times for each component. Note that because X-means expects inputs in vector form, we were unable to apply our direct Levenshtein-distance approach with clustering, but we include some runtimes for FPF Levenshtein to provide a comparison.

That clustering is more expensive and complex than FPF is not surprising; clustering has to perform the additional work of computing clusters, rather than simply ranking items by distance. That FPF produces considerably better discovery curves, as shown in Figures 3–8, *is* surprising. The comparative ineffectiveness of clustering is twofold: the discovery curves do not climb as quickly as with FPF, and (perhaps even more critically) clustering does not ever find all the faults in many cases. In general, for almost all feature sets, clustering over those same features was worse than applying FPF to those features. The bad performance of clustering was particularly clear for GCC wrong-code bugs: Figure 13 shows all discovery curves for GCC wrong-code, with clustering results shown in gray. Clustering at its "best" missed 15 or more bugs, and in many cases performed much worse than the baseline, generating a small number of clusters that were not represented by distinct faults. In fact, the few clustering results that manage to discover 20 faults also did so more slowly than the baseline curve. While GCC wrong-code clustering was particularly bad, clustering also always missed at least three bugs for SpiderMonkey. Our hypothesis as to why FPF performs so much better than clustering is that the nature of fuzzing results, with a long tail of outliers, is a mismatch for clustering algorithm assumptions. FPF is not forced to use any assumptions about the size of clusters, and so is not "confused" by the many single-instance clusters. A minor point supporting our hypothesis is that the rank ordering of clustering effectiveness matched that of the size of the tail for each set of faults: GCC crash results were good but not optimal, SpiderMonkey results were poor, and GCC wrong-code results were extremely bad.

## 5. Related Work

A great deal of research related to fuzzer taming exists, and some related areas such as fault localization are too large for us to do more than summarize the high points.

***Relating Test Cases to Faults*** Previous work focusing on the core problem of "taming" sets of redundant test cases differs from ours in a few key ways. The differences relate to our choice of primary algorithm, our reliance on unsupervised methods, and our focus on randomly generated test cases.

First, the primary method used was typically clustering, as in the work of Francis et al. [9] and Podgurski et al. [28], which at first appears to reflect the core problem of grouping test cases into equivalence classes by underlying fault. However, in practice the user of a fuzzer does not usually care about the tests in a cluster, but only about finding at least one example from each set with no particular desire that it is a perfectly "representative" example. The core problem we address is therefore better considered as one of *multiple output identification* [8] or rare category detection [8, 40], given that many faults will be found by a single test case out of thousands. This insight guides our decision to provide the first evaluation in terms of discovery curves (the most direct measure of fuzzer taming capability we know of) for this problem. Our results suggest that this difference in focus is also algorithmically useful, as clustering was less effective than our (novel, to our knowledge) choice of FPF.

One caveat is that, as in the work of Jones et al. on debugging in parallel [15], clusters may not be directly useful to users, but might assist fault localization algorithms. Jones et al. provide an evaluation in terms of a model of debugging effort, which combines clustering effectiveness with fault-localization effectiveness. This provides an interesting contrast to our discovery curves: it relies on more assumptions about users' workflow and debugging process and provides less direct information about the effectiveness of taming itself. In our experience, sufficiently reduced test cases make localization easy enough for many compiler bugs that discovery is the more important problem. Unfortunately, it is hard to compare results: cost-model results are only reported for SPACE, a program with only around 6,200 LOC, and their tests included not only random tests from a simple generator but 3,585 human-generated tests. In the event that clusters are needed, FPF results for any $k$ can be transformed into $k$ clusters with certain optimality bounds for the chosen distance function [10].

Second, our approach is completely unsupervised. There is no expectation that users will examine clusters, add rules, or intervene in the process. We therefore use test-case reduction for feature selection, rather than basing it on classifying test cases as successful or failing [9, 28]. Because fuzzing results follow a power law, many faults will be represented by far too few tests for a good classifier to

include their key features; this is a classic and extreme case of class imbalance in machine learning. While bug slippage is a problem, reduction remains highly effective for feature selection, in that the features selected are correct for the reduced test cases, essentially by the definition of test-case reduction.

Finally, our expected use case and experimental results are based on a large set of failures produced by large-scale random testing for complex programming languages implemented in large, complex, modern compilers. Most previous results in failure clustering used human-reported failures or human-created regression tests (e.g., GCC regression tests [9, 28]), which are essentially different in character from the failures produced by large-scale fuzzing, and/or concerned much smaller programs with much simpler input domains [15, 23], i.e., examples from the Siemens suite. Liblit et al. [22] in contrast directly addressed scalability by using 32,000 random inputs (though not from a pre-existing industrial-strength fuzzer for a complex language) and larger programs (up to 56 KLOC), and noted that they saw highly varying rates of failure for different bugs. Their work addresses a somewhat different problem than ours—that of isolating bugs via sampled predicate values, rather than straightforward ranking of test cases for user examination—and did not include any systems as large as GCC or SpiderMonkey.

***Distance Functions for Executions and Programs*** Liu and Han's work [23], like ours, focuses less on a particular clustering method and proposes that the core problem in taming large test suites is that of embedding test cases in a metric space that has good correlation with underlying fault causes. They propose to compute distance by first applying fault localization methods to executions, then using distance over localization results rather than over the traces themselves. We propose that the reduction of random test cases essentially "localizes" the test cases themselves, allowing us to directly compute proximity over test cases while exhibiting the good correlation with underlying cause that Liu and Han seek to achieve by applying a fault-localization technique. Reduction has advantages over localization in that reduction methods are more commonly employed and do not require storing—or even capturing or sampling—information about coverage, predicates, or other metrics for passing test cases. Liu and Han show that distance based on localization algorithms better captures fault cause than distance over raw traces, but they do not provide discovery curves or a detailed clustering evaluation. They provide correlation results only over the Siemens suite's small subjects and test case sets.

More generally, the problems of distance functions over executions and test cases [5, 11, 23, 30, 39] and programs themselves [4, 35, 41] have typically been seen as essentially different problems. While this is true for many investigations—generalized program understanding and fault localization on the one hand, and plagiarism detection, merging of program edits, code clone, or malware detection on the other—the difference collapses when we consider that every program compiled is an input to some other program. A program is therefore both a program and a test input, which induces an execution of another program. Distance between (compiled) programs, therefore, is a distance between executions. We are the first, to our knowledge, to essentially erase the distinction between a metric space for programs and a metric space for executions, mixing the two concepts as needed. Moreover, we believe that our work addresses some of the concerns noted in fault-localization efforts based on execution distances (e.g., poor results compared to other methods [16]), in that distance functions should perform much better on executions of reduced programs, due to the power of feature selection, and distances over programs (highly structured and potentially very informative inputs) can complement execution-based distance functions.

***Fault Localization*** Our work shares a common ultimate goal with fault localization work in general [5, 7, 11, 16, 17, 21, 22, 30] and specifically for compilers [43]: reducing the cost of manual debugging. We differ substantially in that we focus our methods and evaluation on the narrow problem of helping the users of fuzzers deal with the overwhelming amount of data that a modern fuzzer can produce when applied to a compiler. As suggested by Liu and Han [23], Jones et al. [15], and others, localization may support fuzzer taming and fuzzer taming may support localization. As part of our future work, we propose to make use of vectors based on localization information to determine if, even after reduction, localization can improve bug discovery. A central question is whether the payoff from keeping summaries of successful executions (a requirement for many fault localizations) provides sufficient improvement to pay for its overhead in reduced fuzzing throughput.

## 6. Conclusion

Random testing, or fuzzing, has emerged as an important way to test compilers and language runtimes. Despite their advantages, however, fuzzers create a unique set of challenges when compared to other testing methods. First, they indiscriminately and repeatedly find test cases triggering bugs that have already been found and that may not be economical to fix in the short term. Second, fuzzers tend to trigger some bugs far more often than others, creating needle-in-the-haystack problems for engineers who are triaging failure-inducing outputs generated by fuzzers.

Our contribution is to *tame* a fuzzer by adding a tool to the back end of the random-testing workflow; it uses techniques from machine learning to rank test cases in such a way that interesting tests are likely to be highly ranked. By analogy to the way people use ranked outputs from static analyses, we expect fuzzer users to inspect a small fraction of highly ranked outputs, trusting that lower-ranked test cases are not as interesting. If our rankings are good, fuzzer users will get most of the benefit of inspecting every failure-inducing test case discovered by the fuzzer for a fraction of the effort. For example, a user inspecting test cases for SpiderMonkey 1.6 in our ranked order will see all 28 bugs found during our fuzzing run 4.6× faster than will a user inspecting test cases in random order. A user inspecting test cases that cause GCC 4.3.0 to emit incorrect object code will see all 35 bugs 2.6× faster than one inspecting tests in random order. The improvement for test cases that cause GCC 4.3.0 to crash is even higher: 32×, with all 11 bugs exposed by only 15 test cases.

## Acknowledgments

We thank Michael Hicks, Robby Findler, and the anonymous PLDI '13 reviewers for their comments on drafts of this paper; Suresh Venkatasubramanian for nudging us towards the furthest point first technique; James A. Jones for providing useful early feedback; and Google for a research award supporting Yang Chen. A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786.

## References

[1] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Proc. ASE*, pages 19–28, September 2008.

[2] Abhishek Arya and Cris Neckar. Fuzzing for security, April 2012. `http://blog.chromium.org/2012/04/fuzzing-for-security.html`.

[3] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *Proc. ICSE*, pages 452–462, June 2012.

[4] Silvio Cesare and Yang Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *Proc. TRUSTCOM*, pages 181–189, November 2011.

[5] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Proc. FSE*, pages 73–82, 2004.

[6] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP*, pages 268–279, 2000.

[7] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. ICSE*, pages 342–351, May 2005.

[8] Shai Fine and Yishay Mansour. Active sampling for multiple output identification. *Machine Learning*, 69(2–3):213–228, 2007.

[9] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *Proc. ISSRE*, pages 451–462, November 2004.

[10] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[11] Alex Groce. Error explanation with distance metrics. In *Proc. TACAS*, pages 108–122, March 2004.

[12] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proc. ICSE*, pages 621–631, May 2007.

[13] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proc. ISSTA*, pages 78–88, July 2012.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proc. USENIX Security*, pages 445–458, August 2012.

[15] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proc. ISSTA*, pages 16–26, July 2007.

[16] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. ASE*, pages 273–282, November 2005.

[17] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE*, pages 467–477, May 2002.

[18] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Proc. SAS*, pages 203–217, September 2005.

[19] Ted Kremenek and Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proc. SAS*, pages 295–315, June 2003.

[20] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[21] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proc. PLDI*, pages 141–154, June 2003.

[22] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. PLDI*, pages 15–26, June 2005.

[23] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Proc. FSE*, pages 46–56, November 2006.

[24] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, December 1998.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI*, pages 89–100, June 2007.

[26] Dan Pelleg and Andrew Moore. Active learning for anomaly and rare-category detection. In *Advances in Neural Information Processing Systems 18*, December 2004.

[27] Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proc. ICML*, pages 727–734, June/July 2000.

[28] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proc. ICSE*, pages 465–475, May 2003.

[29] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proc. PLDI*, pages 335–346, June 2012.

[30] Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE*, pages 30–39, October 2003.

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

[32] Jesse Ruderman. Mozilla bug 349611. `https://bugzilla.mozilla.org/show_bug.cgi?id=349611` (A meta-bug containing all bugs found using jsfunfuzz.).

[33] Jesse Ruderman. How my DOM fuzzer ignores known bugs, 2010. `http://www.squarefree.com/2010/11/21/how-my-dom-fuzzer-ignores-known-bugs/`.

[34] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *CACM*, 18(11):613–620, November 1975.

[35] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. SIGMOD*, pages 76–85, June 2003.

[36] Alexander Strehl and Joydeep Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *The Journal of Machine Learning Research*, 3:583–617, 2003.

[37] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proc. ASE*, pages 253–262, November 2011.

[38] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. ICSE*, pages 45–54, May 2010.

[39] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *Proc. ESEC/FSE*, pages 293–294, August 2009.

[40] Pavan Vatturi and Weng-Keen Wong. Category detection using hierarchical mean shift. In *Proc. KDD*, pages 847–856, June/July 2009.

[41] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, July 2006.

[42] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. ICSE*, pages 461–470, May 2008.

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

[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI*, pages 283–294, June 2011.

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