SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–33
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Using mutants to help developers distinguish and debug (compiler) faults

Josie Holmes[1], Alex Groce[1]

[1]*School of Informatics, Computing & Cyber Systems, Northern Arizona University*

SUMMARY

Measuring the distance between two program executions is a fundamental problem in dynamic analysis of software, and useful in many test generation and debugging algorithms. This paper proposes a metric for measuring distance between executions, and specializes it to an important application: determining similarity of failing test cases for the purpose of automated fault identification and localization in debugging based on automatically generated compiler tests. The metric is based on a causal concept of distance where executions are similar to the degree that changes in the program itself, introduced by mutation, cause similar changes in the correctness of the executions. Specifically, if two failing test cases for a compiler can be made to pass by applying the same mutant, those two tests are more likely to be due to the same fault. We evaluate our metric using more than 50 faults and 2,800 test cases for two widely-used real-world compilers, and demonstrate improvements over state-of-the-art methods for fault identification and localization. A simple operator-selection approach to reducing the number of mutants can reduce the cost of our approach by 70%, while producing a gain in fault identification accuracy. We additionally show that our approach, though devised for compilers, is applicable as a conservative fault localization algorithm for other types of programs, and can help triage crashes found in fuzzing of non-compiler programs more effectively than a state-of-the-art technique.

Received ...

KEY WORDS: program mutation, fuzzer taming, fault localization, compiler testing, automated debugging

1. INTRODUCTION

The proposed techniques in this paper are motivated in large part by progress in automated testing. Recent advances, some implemented in powerful tools [103, 84, 40, 42, 55, 56], have made it easier to find subtle faults in modern optimizing compilers. Unfortunately, the blessing of many failing test cases can become a curse for a developer. Simply discovering which test cases in a large set of failures represent distinct faults (or already known faults) is a very difficult problem [16, 23]. For simple crashes and assertion violations, bucketing failures is sometimes easy. When a failure is detected by differential testing [66], where two compilers or optimization levels disagree, the problem is so hard that it may turn potential users away from automated test generation [16]. Such *wrong code* bugs do not produce a crash, only two compiled programs with different, incorrect (in at least one case) behavior. Even once a fault is identified, fixing the problem can be very difficult. Due to complexity of intermediate representation, interaction of optimizations, subtle language semantics, and other issues, debugging compiler faults may often be harder than debugging faults in other kinds of software (this is probably not limited to compilers; debugging systems software in general is hard [67]). This paper proposes the use of mutants [4] to assist debugging based on large sets of automatically generated test cases. The underlying idea is that if two failures can be repaired, by applying the same program mutant, *even if the repair only avoids the failure and does*

not truly fix it, the two failures are likely due to the same fault. Repairs can also provide information for understanding and debugging discovered faults. The approach suggests a plausible metric for comparing any two program executions, not just failing test cases. Comparing executions has been a prominent concept in dynamic analysis throughout much of the last two decades [82, 90] (e.g., Ball's early discussion of profile similarity [7]).

1.1. Fuzzer Taming and Debugging Assistance

Automated test generators tend to produce many failures, which correspond to far fewer faults. The distribution of faults in a set of failures tends to follow a strong power-law curve, where a few faults account for most failures, and many or even most faults have only 1 or 2 associated failures (even in a set of 1,000 or more test cases) [16, 35]. Looking at all the failures to find distinct faults is impractical. It is also impractical to apply the iterative approach: first, fix one test's fault; then re-run all tests, until all tests pass. There may be some (possibly known) faults that are very difficult to fix. As of 5/8/2017, clang had 107 unresolved, assigned bugs, some dating to 2008 [1]. Knowing if a randomly generated test case is an instance of a known failure can be difficult, and fixing long-standing problems is unlikely. Applying a novel fuzzing technique [103, 40, 55] may dump dozens of new faults on a project at once, and high priority faults may be hidden by stubborn low-priority faults.

Chen et al. defined 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." [16]. Their proposed solution was to use the Furthest-Point-First [25] (FPF) algorithm. An FPF ranking of test cases requires a distance metric d on test cases, and ranks test cases so that dissimilar tests appear earlier. The hypothesis of Chen et al. was that dissimilar tests, by a well-chosen metric, will also fail due to different faults. FPF is a greedy algorithm that proceeds by repeatedly adding the item with the maximum minimum distance to all previously ranked items. Given an initial seed item r_0, a set S of items to rank, and a distance metric d, FPF computes r_i as $s \in S : \forall s' \in S : min_{j<i}(d(s, r_j)) \geq min_{j<i}(d(s', r_j))$. The condition on s is obviously true when $s = s'$, or when $s' = r_j$ for some $j < i$; the other cases for s', however, force selection of some (possibly non-unique) max-min-distance s.

Fuzzer taming results using FPF were promising, but limited. Rather than proposing a universal metric for use in fuzzer taming, Chen et al. offered many possible metrics (based on ideas in the literature, such as edit distances [58] or coverage spectra [83] methods), none of which performed notably well across all three of their benchmark data sets, and some of which were not even applicable to all data sets. For the two most difficult benchmarks, none of the metrics performed extremely well. FPF-based approaches remain in need of a universal but effective metric, particularly for the critical first 50 test cases examined [16].

One reason fuzzer taming is needed is that debugging compiler faults is, as noted, difficult. If developers were able to fix faults quickly once they were identified, it would not be as important to have very good fault identification. Many duplicate failures could be removed by fixing the underlying faults. Ideally, a technique able to provide effective fuzzer taming should be able to translate its underlying rationale into some form of automated debugging assistance. A variety of automated debugging methods [82, 32] have been based on distance, so there is reason to expect that a high-quality metric might also contribute to fault localization.

1.2. Comparison by Response to Mutation

The methods we propose are based on an idea with numerous possible applications: *if two test cases that fail can be made to succeed by the same small modification to the original program P, this provides evidence that they fail due to the same fault*. The idea is hardly controversial: most developers determine if they have fixed all faults in a set of failing test cases by re-running the tests after they fix at least one fault. As a method for fuzzer taming or debugging assistance, however, this is useless. Fortunately, developer-provided patches are not the only source of changes to a program. Mutation analysis [21, 10] applies large numbers of small syntactic changes to a program, usually to evaluate a test suite [4, 52]. Some recent work has also considered these changes as a source of fixes

DOI: 10.1002/stvr

or localizations for program faults [80, 45, 68, 20]. Published empirical data [30], as well as our data, suggests that very few compiler faults are due to the kind of simple syntactic changes found in mutants. However, what should matter is that failures, which correspond to faults, can be made to succeed. When a test case t fails for program P but succeeds for mutant m of P we say that m *repairs* t (in practice, we will usually restrict this to unkilled mutants m). In most cases this does not mean that m is the inverse of a fault, but that m somehow avoids the consequence of a fault. It is not important that repairs actually fix anything, if all we seek is a way to decide whether failures are due to the same fault.

This idea is in fact a consequence of a more general (and difficult to investigate) hypothesis. What is interesting about a program execution? For many purposes, the only interesting properties of an execution are those that in some way relate to the correctness properties of the program, assuming we have a good set of correctness properties. These may include assertions, differential testing [66, 37], timing constraints, temporal logic formulas, or any other checkable properties. What, then, makes two executions similar? Our proposal is that *two executions of P are similar to the degree that changes to P produce the same changes in correctness of the executions.*

This leads to the general idea of comparing any two executions by their response to mutants, since a mutant can cause a passing or failing test to violate more/different properties. This generalization to any two executions may at first glance suggest that our approach subsumes past notions of similarity, such as spectra of code coverage, since if two executions do not cover a statement, they both cannot alter correctness if that statement is altered. The metrics are in fact simply incomparable: for example, differences in coverage that never have an effect on correctness cannot contribute to similarity in our approach; with weak correctness properties most executions are very similar. This is similar to the idea of *checked coverage*: code that cannot impact correctness is not interesting [86]. However, for practical purposes the most interesting implication of our proposal is that if two failing test cases can be made to differ as to whether they fail by applying a program mutant, this provides evidence that they fail due to different faults.

1.3. The Onion-Ring and Causality

This is not a claim that being repaired by the same mutant perfectly predicts fault equivalence. To understand why that seems highly unlikely, consider why a mutant of a compiler might cause a test case to stop failing. Many, likely most, compiler faults are due to incorrect optimizations. Turning an optimization off can cause a program to compile correctly. Optimizations are generally guarded by conditional switches — in some cases, so that the optimization can be turned off by a user, in other cases when a property of the code in question prevents the optimization. If two failures become successes due to turning off an optimization, it is presumably *some* level of evidence they are the same fault, compared to the baseline probability. Intuitively, the evidence provided seems much stronger than the fact that, e.g., the two test cases execute a shared line of code [83]. However, optimization guards range from guarding very specific fragments of code to guarding entire modules with many functions and related sub-optimizations. If a mutant prevents a very specific optimization (consisting of a few lines of code) from executing, and repairs two tests, that is powerful evidence they are related. If a mutant turns off a large number of optimizations, it is weak evidence. Must we then characterize mutant granularity? No.

Consider the structure of mutants repairing a failure due to some optimization. If many mutants repair this failure, by turning off the optimization or a whole family of optimizations, some will likely repair failures due to different faults. Repairs naturally produce an *onion-ring* structure (Figure 1), where a failure is repaired by (1) some mutants that are very specific to its fault (the inner rings) and repair few failures, as well as other mutants (2) that repair more and more faults, up to a hypothetical mutant (3) that turns off all optimization in the compiler (e.g., by effectively adding `-O0`), the outer ring of the onion. This does not introduce confusion, because the more repairs two tests have in common, the more likely they are to be due to the same fault, and the more they disagree on, the less likely they are to be due to the same fault. While especially intuitive as a consequence of disabling compiler optimizations, it seems likely this structure appears for repairs in many programs. Disabling an optimization is a code change that (usually) does not cause any test

DOI: 10.1002/stvr

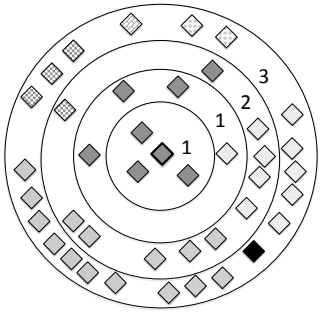

Figure 1. Onion-ring shows all repairs for the center-most test case, and all other test cases repaired by same mutants. Rings represent repairing mutants, and diamonds test cases. Diamond pattern symbolizes fault. For innermost ring, all failures are due to the same fault. As rings expand, the set of faults grows (as does distance).

case that only concerns semantics, not speed, to fail, but can avoid some failures. For some oracles, many changes may fall into this classification: for example, if the only oracle is to detect crashes, a fault can be avoided by aborting the program early. Turning off a data-structure invariant checker is another possible false repair that might cover many different faults. In all these cases, assuming more interesting and specific ways to repair faults exist, a distance metric can still work.

The idea of measuring the distance between failures (or other test cases) by their response to mutants is appealing. The most important aspect of most applications of distances between executions is some aspect of causality [59, 106]. Since at least the work of Hume [46], causality has been understood as answering some question of the form "what would have made a difference in this effect?" A program mutant is a very direct way to answer this question. The problem with determining causes of events in computer programs is that, as in most problems of causality, too many things can make a difference, most of which are not interesting. Program mutants are much more likely to be interesting causes for events in a program than changes to test cases, at least for fault identification and debugging purposes, because the practical use of knowing a cause is to change the cause and avoid or produce an effect. Debugging by changing test cases is not useful; debugging by altering the program is precisely how we seek to use the causes we discover. Using mutants as causes in order to measure similarity "inverts" the ideas of Lewis [59, 60], where similarity determines causality. One pleasant side-effect is that execution representation [82, 83] is not a difficulty for this kind of metric, since an execution is always represented as a set of correctness evaluations.

This paper investigates the utility of using distance metrics based on the causal approach suggested above. We show that such a metric can improve, in a statistically significant way, the ability to distinguish tests based on which faults they trigger. Furthermore, fault localization based on the same metrics can successfully pinpoint the location of some compiler faults sufficiently to be likely to substantially aid debugging, and, when applicable, performs very well for some faults studied in recent papers on mutant-based fault localization [45, 12].

1.4. Contributions

This paper extends our 2018 ISSRE paper [43] introducing our approach. That paper introduced a novel way of measuring distance between program executions, but more importantly provided some practical proposals for using such a metric in compiler fuzzer taming and fault localization. Our experimental results show that, for two realistic complex compilers, Mozilla's SpiderMonkey 1.6 JIT for JavaScript and GCC 4.3.0, using mutant response metrics improves FPF-based fuzzer taming over best previous results [16], in multiple ways. Our fault localization approach also compares favorably to state-of-the-art fault localization techniques [68, 45, 74] for non-compiler programs, albeit with a more limited applicability (but also a lower cost). We extend the earlier paper by:

- generally expanding discussion of our approach, including the conceptual basis for using mutants to measure similarity of executions, and related work,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- providing an extended example (in Section 4) to help readers better understand the metric, FPF-based fuzzer taming, and our fault localization approach,
- introducing and evaluating another compiler fault localization approach (in Section 5.3), based on code coverage changes,
- and including a much more in-depth examination of the computational costs of our approach and various ways of reducing that cost, including experimental evaluations (in Section 6).

2. A MUTANT-BASED DISTANCE METRIC

For every failing test case, its response to mutants defines a bit-vector, with length equal to the number of mutants that repaired any test case, where a 1 bit indicates that the mutant in question repairs this test case*. These cannot be effectively compared with Hamming distance. Consider t_1 and t_2, both repaired by 100 mutants. If these two test cases share 90 common repairing mutants, and disagree for 10, they have a Hamming distance of 20. This makes them "less similar" than two test cases, t'_1 and t'_2, where t'_1 is repaired by 5 mutants, t'_2 is repaired by 5 mutants, and *none of those mutants overlap.* A Jaccard distance, also used (as similarity) in some fault localization algorithms [13], is a better fit. For bitvectors u and v of length n:

$$d(u,v) = \begin{cases} \frac{\sum_{i=0}^{n-1} 1 \text{ if } u_i \neq v_i \text{ else } 0}{\sum_{i=0}^{n-1} 1 \text{ if } u_i \neq 0 \vee v_i \neq 0 \text{ else } 0} & \exists i : u_i \neq 0 \vee v_i \neq 0 \\ 0 & otherwise \end{cases}$$

In other words, the Jaccard distance is $\frac{bitcount(u \veebar v)}{bitcount(u \vee v)}$ (the number of 1 bits in the XOR over the number of 1 bits in the OR for u and v); that is, the distance is the portion of mismatches over repairing mutants for either test. This metric agrees with our intuitions about the above comparisons. It essentially formalizes the intuition of the onion-ring model, where more matching repairs indicates a much higher probability two failures are due to the same fault, even if not all repairs match.

Given such a metric, FPF [25] fuzzer taming [16] begins by ranking as first an arbitrary test case. FPF next ranks the test case, of all unranked tests, that has the largest minimum distance from all ranked test cases — the test case most dissimilar by *closest* already ranked test — until all tests are ranked. FPF is boundedly close to optimal under certain assumptions [25].

More complex metrics are possible, also combining information such as coverage and language features or test case tokens [16]. However, mixtures of metrics (combining different features) other than Levenshteins [58] over test case + output never performed well compared to more focused metrics in earlier work [16]. Therefore, in order to evaluate the contribution of our repair-based metric we compare it, alone, to the previous best metrics.

In Section 10 we discuss uses of d requiring comparison of both failing and passing test cases: in this case, we would use three-valued vectors, with a bit for each *property*, where 1 indicates a failing property succeeds and -1 that a succeeding property fails. For such an application, we could remove the requirement that a repair not be a killed mutant, since the passing test vectors would encode that information

3. LOCALIZATION AND EXPLANATION

Automated fault localization [98, 51] provides users with information about the likely location of a fault in a program's source code. Error explanation [32] also provides information about the causal

*By *repairing* a failing test we here mean that a mutant causes the test to pass, *and* the mutant does not cause other, previously passing tests, to fail.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

structure of a fault — how it can be avoided, what triggers it — a kind of "fault story." Parnin and Orso recently suggested that fault localization methods are not actually providing a great deal of assistance to real users in debugging [75]. Automated fault localization has not been widely adopted in industry.

Parnin and Orso note that most popular methods are statistical and tend to simply provide an ordered list of statements to examine. These statements are not guaranteed to be essentially related to the fault — with many techniques, the statements are, too often, code that executes as a consequence of the fault, code accidentally executed with the fault, and so forth. We note that most evaluations of fault localization have been performed over programs where it is not clear that debugging is extremely hard, vs. the time spent applying automated localization and establishing effective (automated) testing to support automated localization. Compiler debugging, on the other hand, especially for subtle optimization-based problems that produce incorrect code (vs. simple crashes) is generally known to be difficult. Researchers reporting subtle compiler bugs often observe the time from reporting a fault to its correction, even once the fault is assigned, to be lengthy, and the resulting patches are sometimes thousands of lines [16]. Important compilers, whether JITs where security is paramount or C compilers used to compile core systems code, also tend to already be the subjects of extensive automated testing [84, 103].

Repairing mutants can be used for localization and explanation. While few will be equivalent to actual fixes, they will sometimes overlap the incorrect code, and will often be closely related to incorrect code, semantically. Since some tests are repaired by a large number of mutants, it is important to rank the mutants to produce a localization. Based on the onion-ring model proposed above, this is simple: each mutant is ranked as a localization/explanation based on the *most distant failure it also repairs.* If a given mutant only repairs failures that have similar repair vectors to the test case being localized, it will be highly ranked. If it repairs even one very dissimilar failure, then it will be considered a poor localization.

> For a set F of failing test cases and M of mutants, the **Repair** localization for $f \in F$, where $R(f) \subseteq M$ is the set of mutants repairing f, ranks statements by the function $r_f(m)$:
>
> $$r_f(m) = \begin{cases} \frac{1}{1+max_{t \in F : m \in R(t)} d(t,f)} & m \in R(f) \\ 0 & m \notin R(f) \end{cases}$$

This ranks a mutant m (and associated statement) by the inverse of the distance from f to the most distant failing test also repaired by m, adding 1 so that if m only repairs failures at distance 0 r_f is well defined (and maximal). If m does not repair f, it is not ranked at all. One advantage of this approach over some statistical methods is that by ranking *mutants* the localization offers developers not simply a ranked statement but a kind of "explanation" [32] of the fault: if *this mutant* were applied, the fault would not exhibit. As Kochhar et al. show [54], a large majority (more than 85%) of developers considered the ability of a localization tool to provide a *rationale* or reason for a localization important. A mutation plus the tests that stop failing when it is applied is a practical, concrete explanation of a fault localization. Mutants that repair a failure provide insight into the nature of the fault, and, we hypothesize, more information the closer they are to the center of the onion-ring.

In general, we expect **Repair** to be used in the context of compiler (or other complex system) fuzzing, where there are numerous failing tests, likely from multiple faults. In such cases, the f to be localized is presumably selected by FPF. However, in case our localization approach is used in a setting where single faults are expected, we propose an alternative method: let f be an arbitrary test with as few repairing mutants as possible, to allow the distance metric to focus attention on a few key mutants.

An advantage of this method over some statistical approaches to localization is that it is not potentially confused in settings with multiple faults. Each localization/explanation is based on a single failing test case, which in most cases fails due to only one fault. In fact, other faults (and their failures) theoretically *help* **Repair** rank the mutants.

DOI: 10.1002/stvr

```
01  void compile(program: feature list, optimizeLevel: int) {
02    internalRepresentation = initInternalRep(program);
03    if optimizeLevel > 0: {
04       if feature1 in program:
05          if feature2 in program:
06             corrupt(internalRepresentation);
07       if optimizeLevel > 1: {
08          if feature3 in program:
09             if feature4 not in program:
10                corrupt(internalRepresentation);
11          if feature5 in program:
12             if feature6 in program:
13                corrupt(internalRepresentation);
14       }
15    }
16    /* Will produce wrong code if corrupt called */
17    generateCode(internalRepresentation);
```

Figure 2. A toy "optimizing compiler" with multiple bugs

```
T1: (1, 2)
T2: (3)
T3: (5, 6)
```

Figure 3. Failing tests for the toy compiler.

4. A SIMPLE EXAMPLE

To better understand the causal approach taken in this paper, it is useful to consider a simplified, abstract example, and understand how the approach proposed would apply to this example.

Consider the pseudo-code "compiler" in Figure 2. We can imagine that there is a large amount of code (not shown) in the `initInternalRep` and `generateCode` functions. Assume, however, that the only bugs in the compiler are the `corrupt` calls. Traditional Spectrum-Based Fault Localization (SBFL) techniques can be easily "confused" in multiple-fault settings, and by cases such as this, where there may be strong correlations between code covered by the (fault-free) `initInternalRep` and `generateCode` functions and the features contained in programs, making lines of code far from the fault locations appear suspicous.

Assume that all of our test cases run the compiler with `optimizeLevel` of 2, so as to potentially apply all optimizations. A popular way to fuzz a compiler is to compare program behavior when code is compiled with all optimizations with program behavior when code is compiled with no optimizations, so this reflects real-world compiler testing. A set of delta-debugged failing tests, then, might be those shown in Figure 3.

Further, let us only consider two kinds of mutants: condition-negation and statement-deletion mutants. Obviously, negating the condition on line 3 will make all of our failing test cases pass, and negating the condition on line 7 will make test cases 2 and 3 pass, but not test case 1. Negating the conditions on lines 4-5, 8-9, and 11-12 will repair our test cases, but these changes are also likely to make some passing test case fail (they represent preconditions for compiler optimizations), so these mutants will not be used in our analysis. Finally, the only statements that can be deleted without breaking the code are deletions of the `corrupt` calls on lines 6, 10, and 13. We can identify these mutants, which will form the basis of our distance metric and our fault localization, by the line number of the change, since there is only one mutant per line in the set. Each test can then be associated with a bitvector of

DOI: 10.1002/stvr

5 elements: `(repaired by line 3, repaired by line 6, repaired by line 7, repaired by line 10, repaired by line 13)`. The vectors for our tests, then, are:

```
T1: 11000
T2: 10110
T3: 10101
```

T1 thus has a distance of 0.75 from T2 and T3. T2 has a distance of only 0.5 from T3, however.

4.1. Fuzzer Taming for the Toy Compiler

If the Furthest-Point-First algorithm starts from T1 it will arbitrarily select either T2 or T3 next, then the other test. If it starts with T2 or T3, it will select T1 next, then the other of the two closer tests. Of course, taming a set of three failing tests is not difficult! However, if we imagine a much larger set of failing tests, which contain irrelevant features that cannot be removed by delta-debugging without causing the programs to become syntactically invalid, so long as the basic structure of the cause for failure remains the same, the the FPF algorithm will still select tests with the same bitvectors as T1, T2, and T3. Concretely, if we add these tests (shown with associated bitvectors):

```
T4: (1, 2, 7)     11000
T5: (3, 6)        10110
T6: (1, 5, 6)     10101
T7: (1, 2, 5, 6) 10000
```

then FPF from T1 will proceed to a maximally distant test, such as T5, at distance 0.75. The next ranked test could not have been T4 (distance 0.0) or T7 (distance 0.5), but could have been T2, T3, or T6. After T1 and T5, the next test in the ranking will be the test whose nearest neighbor among already chosen tests is furthest away. The distances to the two tests already chosen are:

```
                          11000  10110
T2: (3)           10110   0.75   0.00
T3: (5, 6)        10101   0.75   0.50
T4: (1, 2, 7)     11000   0.00   0.75
T6: (1, 5, 6)     10101   0.75   0.50
T7: (1, 2, 5, 6) 10000   0.50   0.67
```

The FPF algorithm could choose either T3, T6, or T7 all of which have a nearest neighbor at distance 0.5. Let us assume it chooses T6. At this point, the three distinct bugs are all ranked. However, there is one interesting pattern remaining. The distances to tests are now:

```
                          11000  10110  10101
T2: (3)           10110   0.75   0.00   0.50
T3: (5, 6)        10101   0.75   0.50   0.00
T4: (1, 2, 7)     11000   0.00   0.75   0.75
T7: (1, 2, 5, 6) 10000   0.50   0.67   0.67
```

The algorithm is forced to choose T7, the only remaining test that does not have the same bitvector as another test. T7 is worth ranking, as it shows that a program can be written that will result in wrong code unless both bugs it triggers are fixed.

4.2. Fault Localization for the Toy Compiler

Now let us consider the **Repair** fault localization algorithm applied to this example. If we are trying to fix the problem exhibited by T2, what will the system suggest to us as the location of the fault? Recall that 1) in **Repair**, a localization is the location (and content) of a repairing mutant.

DOI: 10.1002/stvr

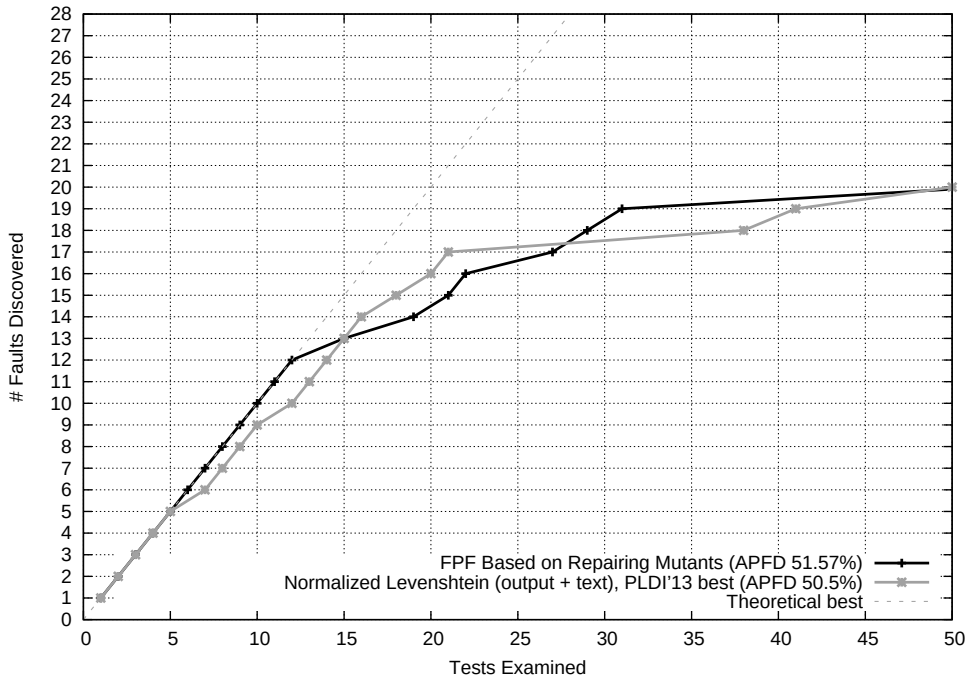

Figure 4. Discovery curves for SpiderMonkey 1.6 faults

T2 is repaired by negating the condition on line 3, negating the condition on line 7, or deleting the statement on line 10. Which of these is best? **Repair** distinguishes between repairing mutants by ordering them by their distance to the most-distant (by bitvector) test also repaired by that mutant. The only other tests repaired by deletion of line 10 are the other instances of the same bug, all at distance 0. Of the other two repairs, negating the condition on line 3 also repairs T1, at distance 0.75, while negating the condition on line 2 repairs nothing farther away than T3 at 0.5. Therefore, the localization for T2 is (in rank order):

```
1: DELETE 10 corrupt(internalRepresentation);
2: NEGATE 07 if optimizeLevel > 1: {
3: NEGATE 03 if optimizeLevel > 0: {
```

4.3. The Onion Ring

In this simple toy example, the onion ring will only have a few layers, one for each possible distance from the test at the center. For example, if we take T2 as the center, then the layers will be:

```
0.00:  T2 T5
0.50:  T3 T6
0.67:  T7
0.75:  T1 T4
```

5. EXPERIMENTAL RESULTS

DOI: 10.1002/stvr

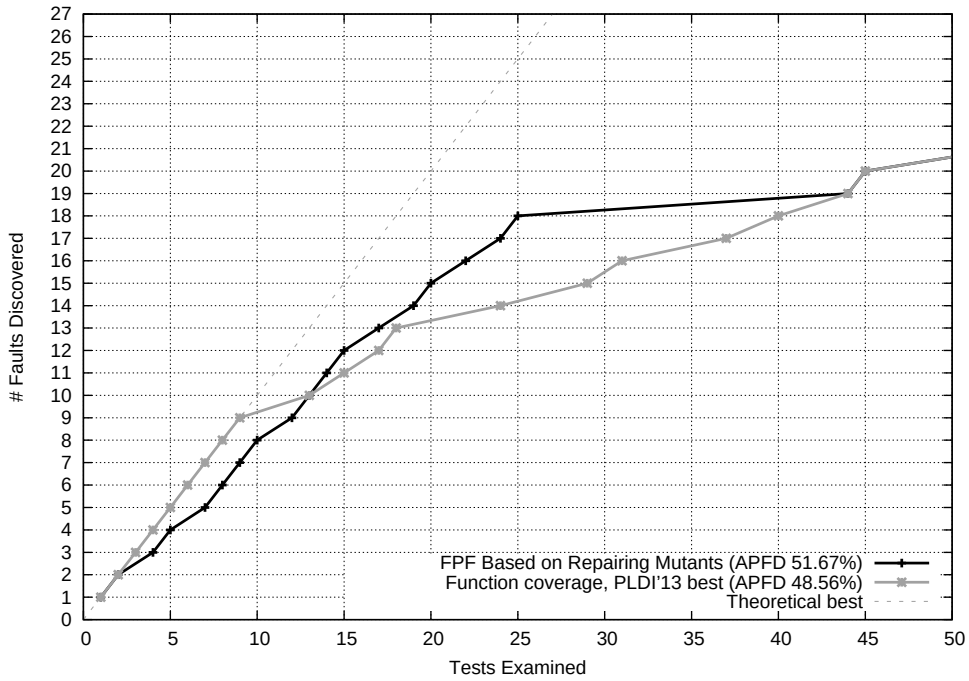

Figure 5. Discovery curves for GCC 4.3.0 wrong-code faults

Our primary experiments are based on subjects used in the only previous study of compiler fuzzer taming (from Chen et al.'s PLDI 2013 paper [16]), which we hereafter refer to as the FPF benchmark. Of the three data sets examined in that paper, this paper considers two: faults and tests cases for SpiderMonkey 1.6, Mozilla's JavaScript engine, with tests generated by `jsfunfuzz` [84], and *wrong code* faults and test cases for GCC 4.3.0, with tests generated by Csmith [103]. GCC 4.3.0 crash faults were essentially perfectly localized by previous approaches. In fact, on examination, only two of the 11 crash bugs are not distinguished by simply examining the crash message. Faults that produce a crash are also likely to be much easier to debug than semantic problems such as all of the GCC wrong code bugs and most of the JavaScript faults– e.g., following a dynamic slice might well suffice.

Our mutants were produced using the tool written by Jamie Andrews [4]. Andrews' tool applies only four operators: statement deletion, conditional negation, operator replacement, and constant replacement, chosen as a small set that still produces good results for C code. Experiments were performed on a MacBook Pro with 16GB RAM and dual-core 3.1GHz Intel Core i7; GCC executed on a VirtualBox-hosted Ubuntu 11.04. For GCC, some test cases from the FPF set no longer failed, presumably due to unknown differences in execution environment, OS, or memory layout. Discarding these reduced the set of test cases from 1,275 to 1,117 and the number of distinct faults to 27 rather than 35. For SpiderMonkey, all 1,749 test cases from the FPF benchmark failed in the new environment, representing 28 distinct faults.

These data sets, though similar in that both compilers are written in C, provide some interesting variance for testing our metrics. The oracle for SpiderMonkey executions is the set of checks built in to `jsfunfuzz` plus the requirement that the execution not crash. This is only a moderately strong oracle, and allows serious deviations from both the JavaScript language specification and normal SpiderMonkey behavior, while checking some complex details, such as `eval` round-trips. For GCC,

DOI: 10.1002/stvr

the oracle is a differential check on a hash code: failure was defined as producing a executable with the -O3 flag that, when executed, produced a different checksum than code compiled by either of GCC 4.9.3 or clang 7.0.0, both using -O0. A repairing mutant must enable GCC 4.3.0 to actually compile code correctly, which is a very strict correctness property, making coincidental correctness [70] highly unlikely—only missed optimizations are allowed.

We only show results for the first 50 tests in the ranking, and computed areas under curves for the same limit, since it seems very unlikely that a user will examine many more than 50 tests, especially given the decreasing slope of the discovery curves. In practice, after 50 tests, fixing a few faults and then re-running tests and FPF seems the most likely recourse, and we confirmed that after removing random subsets of faults, our metrics still outperform the previous best. We applied X-means [76] in an attempt to use clustering with our metrics, but, as with previous results [16] it did not compare well with FPF, and the runtime was much higher. We confirm the conclusion [16] that clustering (at least with X-means) does not work well in a setting with extreme disparity in instance counts for faults. A further difficulty for our setting is that X-means and most off-the-shelf clustering tools take vectors, not a distance metric. X-means is using the raw repair data, not weighted by the fact that 0-0 agreement is much less informative than other possible matchings (a Euclidean over 0-1 vectors is just the square root of Hamming distance).

5.1. Mozilla SpiderMonkey 1.6 Results

There were 96,828 SpiderMonkey mutants, based on 69,634 lines of code in C and header files. Of these mutants, 12,666 were covered by some failing test case. Of these mutants, 10,525 survived a basic set of SpiderMonkey quick tests [33]. Of these mutants, 1,326 (12.6%) repaired at least one test case. Figure 4 shows the discovery curves for the mutant repair metric compared to the ideal discovery curve and the best curve from previous work using FPF, which used a normalized Levenshtein distance [58] over the failure output and test case text [16]. A discovery curve is a plot of the number of distinct faults that a user, examining the tests in the ranked order, would have seen after N tests (here, N goes up to 50).

The APFD (Average Percent Faults Detected) values in the graphs are based on the measure introduced by Rothermel et al. for evaluating test case prioritization methods [22]. APFD is a somewhat better summary of results than the raw curve areas used for evaluation in previous work [16]. APFD, as the name suggests, measures the percent of all faults discovered at the "average" point on the curve, by comparing the curve's area to an ideal curve, with interpolation. A simpler (but less informative) way to compare the curves is to note that the mutation-based metric's curve is above the best previous curve at 60% of data points[†].

APFD results are useful summaries, but the curve itself is also worth examining, since (1) long sequences of tests with no new faults may discourage users more than an overall less effective but steadily climbing curve with few "plateaus" as we call these uninformative sequences of tests and (2) a good early curve is important to developers. The mutant repair curve climbs very rapidly in the early portion, with perfect discovery for the first 12 faults. The largest plateau is 5 tests without a new fault, ignoring the long period after the 31st test. In fact, a plateau at the end of the curve is not problematic. Assuming that few users will examine more than 50 test cases without fixing some faults, the user may give up after seeing 10 or more tests without a new fault based on the mutant repair curve, and in fact lose nothing by doing so. The best FPF benchmark curve, in contrast, has a plateau of size 17 after the 17th fault. If we assume a user stops examining tests after a size 10 or greater plateau, the user will only see 17 faults using the best benchmark metric, vs. 19 with our metric. Stopping at a plateau of size 6 produces the same results.

Over all mutants, and all pairs of test cases repaired by the same mutant (so the same pair may count many times, if many mutants repair both test cases), the probability of being due to the same fault was 42.77%, and the probability of being due to the same fault if two test cases disagreed on

[†]The details of APFD calculation are slightly involved, and the interpolation and perfect curve are not exact fits for fuzzer taming; nonetheless the basic method summarizes curves fairly well and is standard in the testing literature for similar problems [108].

DOI: 10.1002/stvr

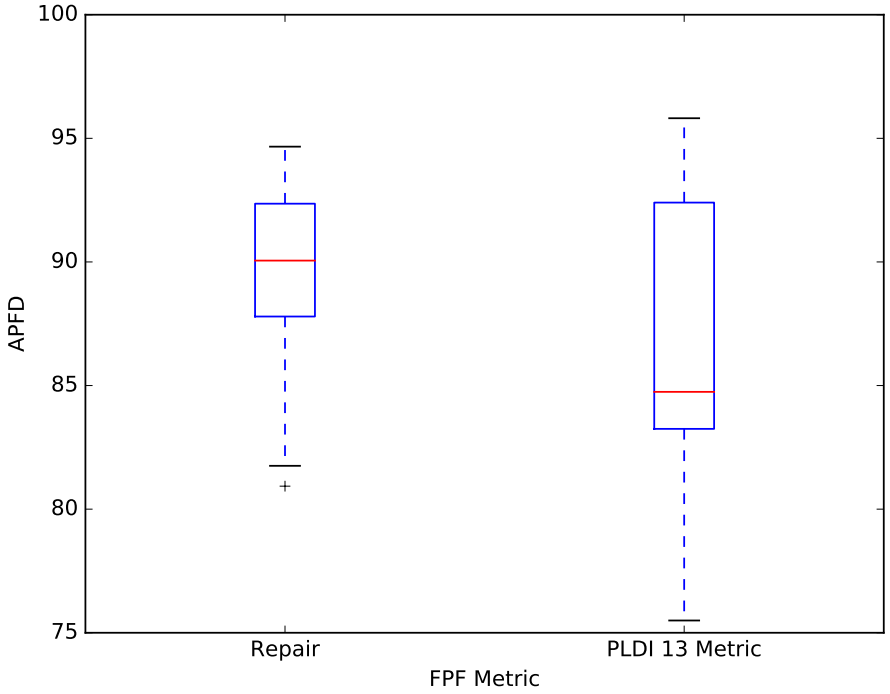

Figure 6. APFD values for Spidermonkey suite slices

a repair (the mutant repaired one test case but not the other) was only 20.33%. The baseline rate for same-fault for test pairs was 33.64%. Just knowing that two test cases have *one* shared repairing mutant makes it 1.27x more likely that they are the same fault, and knowing they differ for one mutant makes it 0.6 times as likely they are due to the same fault. Matching non-repair, however, provides very weak evidence: only a 34.14% chance of matching fault, just over baseline.

An additional measure of effectiveness is to consider how effective a metric is in producing matched nearest neighbors. That is, how often is the nearest neighbor of a failure (that is not due to a singleton fault—a fault detected by only one test case) due to the same fault? For reduced [107, 16, 81] test cases, we assume that for a perfect metric, the nearest neighbor should almost always share the same fault, since there is little or no extraneous semantic content to each test beyond the cause of failure. For the SpiderMonkey failures, 96.3% of non-singleton failures matched their nearest neighbor(s). For mutants that repaired any failures, the mean and median number of test cases repaired were 120.6 and 4.0, respectively. Most mutants repaired a small number of tests, but a few mutants repaired a very large number of tests. A few mutants repaired *all* SpiderMonkey faults; obviously these mutants were not actual fault fixes, but effectively disabled the mechanism `jsfunfuzz` used to detect failure. The mean and median homogeneity for repairing mutants (% of failures repaired corresponding to the most common fault repaired) were 77.96% and 86.36%, respectively.

In order to check our results statistically, we sliced the FPF benchmark tests into 20 randomly selected equal-sized (as much as possible) distinct subsets (Figure 6) . The repair metric had a mean APFD (89.23%) for these subsets that was 2.85% better than the mean APFD for the best benchmark metric (86.76%), over sets of ~ 10 faults. The result was statistically significant ($p < 0.05$ by Wilcoxon test). Median repair APFD was 90.05%, vs. 84.75% for the best benchmark metric.

5.1.1. Fault Localization Table I shows a comparison of three fault localization methods for the SpiderMonkey (and GCC) faults. The first column is a fault ID. The next three columns show

DOI: 10.1002/stvr

localization rankings. A ranking of 1 means that a code location *contained in* the actual faulty code locations was presented to the user as the most likely location of the fault by the method: the method has precisely identified the fault location. A ranking of, for instance, 22, on the other hand would mean that a user examining code locations proposed as faulty by the method would not reach any faulty code until examining the 22nd such location proposed by it. A dash for a column means that localization did not rank any faulty statements, or assigned all faulty statements suspiciousness of 0.0. The three rankings are: our **Repair** localization, the MUSE [68] localization, and the MUSEUM [45] localization. MUSEUM uses the same formula as MUSE, but works better for multiple faults because it uses only one failure. The MUSE/MUSEUM formulas normally make use of information from passing tests as well: when mutating a statement makes a passing test case fail, it makes the statement *less* suspicious, by a weighted amount. The weight assigned to information from passing tests in our setting would likely be low (due to the ratios of repairs to mutation kills). In a limited sense, information from passing tests is already incorporated in our results. Throwing out all mutants that are killed by any passing test as potential repairs/localizations ensures that the rankings of all statements that are in MUSE/MUSEUM rankings are correct, *relative to each other*. By definition, passing tests have no influence on the suspiciousness of these mutants. However, there may be other mutants that 1) repair a failure and 2) are killed by some test case: these could, if enough different ones repaired the same faulty statement, improve MUSE/MUSEUM results, though in practice this is extremely unlikely. We reject such mutants in part to keep costs low, and in part because we think that the causal information contained in such mutants, that "fix" a failure but also break some passing test(s), is problematic. They seem likely to be less useful as explanations of the causes of a failure, since they do not impact the program semantics in a way that is only known to be beneficial, and could potentially even mislead a user if used as explanations.

Discussion with the MUSE/MUSEUM authors confirms that adding information for passing tests, while costly, would likely improve the MUSE and MUSEUM results, given the weakness of our oracles, particularly for SpiderMonkey, and should be considered essential for ideal application of their approach. Because the cost of recording the full mutant analysis matrix for all passing tests (rather than considering a mutant killed as soon as one test fails, as we do) is very high, and we would like to produce a comparison over a fixed computational cost, we give results for MUSE/MUSEUM over failing tests only, cautioning that we are not sure what the impact of this choice is on faults that were not localized. For similar reasons of keeping costs low, we also used the mutation operators of Andrews vs. the more extensive Proteum [64] operators used in MUSE/MUSEUM's evaluations. Even with these restrictions, running all mutants on passing tests until at least a single kill was observed took about 3 times as long as the search for repairs, which we already consider a potentially problematic cost. In Section 5.4 we provide some comparisons with MUSEUM and MUSE fault localization in the context of a full mutation result matrix using their preferred set of operators.

Results are reported as absolute ranks, rather than more sophisticated [68] localization evaluations because our localizations do not assign suspiciousness scores, only rankings, and in accord with the proposal of Parnin and Orso that only localizations ranking a fault in the top few statements may be useful to users [75]. We expect users to examine the mutants and consider how they help avoid the fault. Blind, unaided pointers to portions of compiler code without more semantic information about why the statement is suspicious seem unlikely to work well, based on our experience with complex compiler faults, and the results of Parnin and Orso [75].

Table I only includes 8 of the 28 SpiderMonkey faults. This is due to a problem with the data set: producing ground truth patches for a large application, in the sense of finding a minimal, clearly fault-fixing code change that can be back-ported to the original code is difficult [16]. While we believe the 28 faults identified in the FPF benchmark data are correct, it is very difficult to produce a valid patch of version 1.6 that captures the fix for most of these faults. In some cases the final commit that caused tests to stop failing appeared to only be the end of a complex series of changes that converged on a correct fix. In other cases, the code was modified so extensively before the fix that identifying "the incorrect part" of the original code seemed to owe more to guesswork than certainty. The evaluation of localization therefore only examines the 8 faults for which we could be reasonably certain that the patch to 1.6 was correct and characterized the fault in question

DOI: 10.1002/stvr

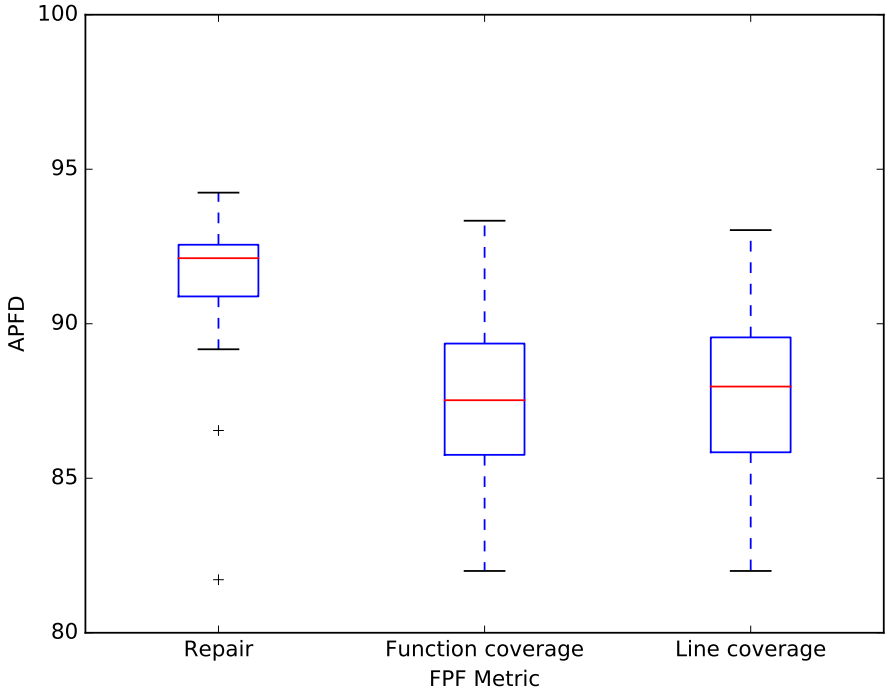

Figure 7. APFD values for GCC suite slices

accurately. In no case was the patch equivalent to one of the mutants (the smallest patch modified two statements).

No method performs extremely well—for half the faults, no methods produced what we consider a useful localization. Compiler faults are hard to debug, and any help is useful, but it seems unlikely developers will really benefit from a localization when it does not rank at least one faulty statement in the first 25-30 statements. By this measure, MUSE only provides a helpful localization once, and performs worst of the methods. **Repair** and MUSEUM both perform well, with **Repair** slightly better.

5.2. GCC 4.3.0 Wrong Code Results

There were 377,679 GCC mutants, based on 424,186 lines of code in C and header files. Of these mutants, 73,016 were covered by some failing test case. Of these mutants, 41,385 survived a GCC bootstrap build, compiled hello world, and passed a GCC test suite. Of these mutants, 3,232 (7.8%) repaired at least one test case. Figure 5 shows discovery curves for the mutant repair metric vs. the best benchmark curve, Euclidean distance over function coverage vectors [16]. GCC's APFD improvement is larger than that for SpiderMonkey, but its early curve (first 15 tests) is worse compared to the benchmark curve, but still has a maximum gap of only 2 redundant tests vs. 4 for the benchmark. Over all, again the curve is better than the best benchmark curve at 60% of all points. Using the hypothetical model where a user stops examining tests after a plateau of size 10, the user of the mutant-based taming will see 18 distinct faults by examining 25 tests, and the user of the function coverage will see 20 faults after 45 tests. If a user gives up after a plateau of size 5, our approach again lets a user examine 18 faults over 25 tests. Function coverage only yields 16 faults over 31 tests.

For GCC wrong code faults, the baseline chance that two test cases shared an underlying fault was 38.97%. Knowing that two test cases shared a repairing mutant raised the chance to 68.5%, and

DOI: 10.1002/stvr

Table I. Fault localization for compiler faults

SpiderMonkey				GCC			
ID	Repair	MUSE	MUSEUM	ID	Repair	MUSE	MUSEUM
1	-	-	-	1	-	157	-
2	-	173	-	2	1	99	14
3	1	22	4	3	-	146	-
4	-	-	-	4	-	-	-
5	19	118	26	5	-	109	-
6	28	87	23	6	-	-	-
7	-	-	-	7	-	-	-
8	3	133	5	8	4	158	18
				9	-	-	-
				10	-	-	-
				11	10	167	6
				12	1	172	195
				13	-	-	-
				14	34	170	125
				15	5	159	31
				16	-	-	-

knowing they disagreed on a repair lowered it to only 19.01%. The respective increase and decrease in probability of matching fault compared to baseline was thus 1.75 times greater for matching repairs, and less than 0.5 the chance of being the same fault, given one mismatched repair. Knowing two test cases were both not fixed by a mutant again provided marginal evidence of sharing a fault: 39.2% chance of matching faults. For 99.3% (all but 8) of the 1,090 non-singleton fault failures in GCC wrong code, the closest test case(s) by the repair metric showed the same fault. The best previous reported FPF metric, function coverage vectors, had a matching rate of 92.2%. The mean and median numbers of repaired tests per mutant (for mutants that repaired any tests at all) were 18.4 and 3.0, respectively. A few mutants fixed a very large number of tests, up to 1,050. These all appear to be turning off all optimizations—essentially running gcc in -O0 mode. Most mutants fixed only a few tests, to a greater extent than was true with SpiderMonkey. Mean homogeneity was 79.4% (median 100%).

Slicing the GCC tests into 20 random equal-sized test subsets and comparing with function and line coverage metrics (Figure 7) we find that differences in APFD values are statistically significant between our metric and both benchmark metrics, by Wilcoxon test, with $p < 0.0005$, and the mean APFD improvement—even for only 55 tests, exposing only 6-11 faults (mean of 7.75 faults)—is more than 3.8% better than either benchmark metric.

5.2.1. Fault Localization Table I shows localization results for GCC 4.3.0. Again, only some faults were deemed to have strong enough ground truth patches for evaluation. MUSE performed poorly, with no useful (by the standard of having at least one faulty statement in the top 25) localizations. MUSEUM provided 3 useful localizations, but no very high quality localizations. Only **Repair** performed very well, with useful localizations for 5 faults, and the fault was in the top 5 statements for 4 of these.

5.3. Coverage Localization

While **Repair** is often a high-quality localization technique, it often fails to localize a fault at all. This failure suggested that for the compiler setting, we should consider other localizations methods, even ones we do not consider likely to be helpful in more general settings. Compiler faults are often so complex and hard to debug that even a somewhat informative localization might be useful.

By definition, when a mutant repairs a test case, it changes the behavior of the test. In some cases, this behavioral change may be limited to changes in program state and data values. However, the vast majority of mutant repairs *for a compiler* can be expected to also modify the code coverage of a test; in our compiler experiments, there were no repairs that did not change coverage; this would

Prepared using stvrauth.cls
DOI: 10.1002/stvr

be very unlikely to hold in other software settings, where many bugs related to data values; for well-fuzzed compilers, however, few bugs are simple, e.g., off-by-one bugs in an integer value. If a fault can be avoided by disabling a problematic optimization in a compiler, this naturally leads to changed coverage. If the fault is due to a bad conditional, where a program path should include additional code, the repair will often change the execution to include the omitted code. As a localization for a fault, then, we can examine the source statements whose coverage changed most often in repairs. Early experiments with this approach showed that a useful coverage change localization must account for the frequency of a change. Some coverage changes are extremely common across all failures, faults, and repairing mutants, and therefore likely not related to any specific fault.

> The **Coverage** localization for a failing test case $f \in F$ (F is the set of all failing test cases) ranks statements by the function r (for rank) where:
>
> $$C(s) = \sum_{t \in F} (\sum_{m \in R(t)} c(m, t, s))$$
>
> $$r(s, f) = \sum_{m \in R(f)} \frac{1}{C(s)} c(m, f, s)$$
>
> $c(m, t, s)$ is 1 if coverage of s changes for test t with mutant m, and 0 otherwise. $R(f)$ is the set of mutants repairing f.

That is, the ranking is based on the number of times a statement's coverage changes its value (covered or not covered) from the original execution of f in executions of mutants that repair f. Changes are weighted by their inverse frequency over all failures and all mutants. Higher values for r result in a higher ranking (more likely to be faulty).

The coverage localization also provides a kind of explanation, though a less direct one than with a repair. For any interesting line of code in the coverage-change-based localization, the user can investigate a particular instance of the change, by selecting a single mutant and stepping through the execution of the failing test case in a debugger to the point of change. The user can also browse through all mutants that produce this coverage change for the test case.

Figure 8 shows part of the patch for SpiderMonkey fault 115, and the **Repair** and **Coverage** outputs that successfully localize the fault (the 3rd mutant output, and the 6th coverage change output), to give some idea of the information our methods present. For **Coverage**, it is surprising that the sixth highest ranked change only appeared in 5 of 148 repairs. This change ranked high out of thousands of changes because it was unusual, appearing for only 10 repairs in the entire SpiderMonkey analysis.

Table II compares the **Coverage** localization with the **Repair** localization. In general, obviously, **Coverage** is not as effective as **Repair**. In fact, in cases where both methods provide any localization information, there is only one instance where **Coverage** is better than **Repair**. On the other hand, **Coverage** provides a localization for 10 faults where **Repair** offers no assistance. The rankings for these suggestions are often poor (indeed, in some GCC cases, they are almost ludicrously bad). However, for GCC faults 10 and 13, **Coverage** produced a useful localization when all of the other methods (including MUSE and MUSEUM) failed to localize at all. Because compiler faults are so difficult to understand and fix, it might be useful for developers to examine the top few localizations (that differ) for multiple methods. While MUSE and MUSEUM both generally perform better than **Coverage**, they almost never provide better information than **Repair**. In every case where MUSE or MUSEUM provides a useful localization (within top-25) and performs better than **Repair**, the difference in ranking between MUSE or MUSEUM and **Repair** is five or fewer ranking positions. **Coverage** uses a sufficiently different approach that it offers potentially useful information, even when **Repair** performs badly.

5.4. Repair Localization for Non-Compiler Faults

The **Repair** localization was designed for use in compiler (or at least complex system software) fuzzing. It is not proposed (or at least, not here evaluated) as a general-purpose fault localization

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Diff of old (<) vs new (>) for SpiderMonkey fault 8 patch (portion):

```
...
<              vp[1] = OBJECT_TO_JSVAL(thisp);
<          } else {
<              ok = OBJ_GET_PROPERTY(cx, thisp, id, &v);
<          }
...
<                      a->avail = (jsuword)sp;
<                }
...
---
>          RESTORE_SP(fp);
...
```

Mutant information:

```
Failure repaired by 148 mutants.
...
#3: delete statement mutant of jsinterp.c:944:
             vp[1] = OBJECT_TO_JSVAL(thisp);
         } else {
/*MUTANT     ok = OBJ_GET_PROPERTY(cx, thisp, id, &v);*/
         }
```

Coverage change information:

```
...
#6: 5 mutants (3.38% of repairs) added jsinterp.c:992:
                    a->avail = (jsuword)sp; /* ADDED */
```

Figure 8. Patch and explanation for fault 115

SpiderMonkey			GCC		
ID	Coverage	Repair	ID	Coverage	Repair
1	2,255	-	1	44,561	-
2	720	-	2	75	1
3	190	1	3	-	-
4	-	-	4	3,067	-
5	18	19	5	186	-
6	46	28	6	490	-
7	-	-	7	-	-
8	6	3	8	260	4
			9	465	-
			10	19	-
			11	43,903	10
			12	85	1
			13	2	-
			14	302	34
			15	223	5
			16	639	-

Table II. Coverage localization results

method. However, it is possible to compare **Repair** with other methods, using their full mutation and experimental set, to provide a more generalizable (and fair) comparison than the evaluation vs. MUSE and MUSEUM over compiler faults. We obtained the data from two evaluations of mutant-based fault localization methods: the set of faults used to evaluate MUSEUM [45], and a set of CoREBench [8] faults used in an evaluation of MUSE and Metallaxis-FL by Chekal et. al [12] (we were unable to obtain the original MUSE dataset, thus far). These data sets are not ideal for evaluating **Repair**: for all but 10 of the faults (all from the CoREBench set) there is only a single failing test case in the data. **Repair** for single failing tests is essentially MUSEUM without use of passing tests, except to reject any mutation that breaks a passing test. **Repair** is meant to operate in the case where a program either has numerous failing tests associated with open bugs (e.g., a typical production compiler or complex system software program) or where a program has just been fuzzed for the first time, so has numerous failures representing different faults. The distance metric lets **Repair** make use of these other faults, by enabling it to focus on repairing mutants most relevant

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Fault localization for CoREBench faults

Fault	#Failing Tests	Repair	Stmts	Metallaxis-FL	MUSE
Coreutils 1	2	1	-	1	1
Coreutils 2	3	-	**0**	86	186
Coreutils 3	1	25	-	47	27
Coreutils 4	1	-	**0**	28	431
Coreutils 5	5	-	2	4	7
Coreutils 6	1	-	1	5	206
Coreutils 7	1	1	-	5	9
Coreutils 8	1	-	1	22	160
Coreutils 9	5	-	1	10	157
Coreutils 11	1	-	1	2	1
Coreutils 12	2	-	**0**	6	9
Coreutils 13	1	-	**0**	905	833
Coreutils 14	1	14	-	19	11
Coreutils 15	2	1	-	1	6
Coreutils 16	1	-	**0**	569	447
Coreutils 17	3	-	2	37	195
Coreutils 18	1	-	3	27	3
Coreutils 19	1	1	-	3	1
Coreutils 20	7	1	-	12	3
Coreutils 21	2	-	**0**	20	191
Coreutils 22	2	1	-	6	2
Findutils 27	1	-	9	12	6
Findutils 32	1	-	1	73	70
Findutils 33	1	-	3	10	30
Findutils 35	1	1	-	1	1
Findutils 36	1	9	-	9	19
Findutils 37	1	11	-	22	34
Grep 46	1	-	2	7	10
Grep 47	1	4	-	4	4
Grep 48	1	-	**0**	26	497

to the failing test. **Repair** can, in many cases, have a lower cost than MUSEUM or Metallaxis-FL because it does not need to run mutants that do not repair any failing tests on the passing tests: analysis can begin with failing tests and only analyze passing tests covering repairing mutants.

5.4.1. MUSEUM Results For the six faults used in the original MUSEUM paper (all with a single failing test), **Repair** gives the same (perfect) localization for three faults. One of the other three faults (Bug2) has no repairing mutants (that do not break some passing test) so **Repair** provides no localization information at all. For the remaining two faults, Bug5 and Bug6, **Repair** gives best rankings of 12 and 9, respectively, compared to 8 and 3 for MUSEUM.

5.4.2. CoREBench Results For the full data set in the paper [12], **Repair** does not perform as well as MUSE and Metallaxis-FL, as expected. It is only able to rank a faulty line of code for 12 of 30 faults. However, for these faults, it performs very well indeed, perfectly localizing 7 of the 12 (vs. 3 total perfect localizations for Metallaxis-FL and 4 for MUSE). **Repair** is uniquely best for 5 of the 12, and is tied for best for another 6; for the remaining fault, it performs better than Metallaxis-FL but worse than MUSE. Table III shows full results. The fourth column shows, for those bugs where **Repair** did not rank any faulty statement, how many statements **Repair** proposed for a user to examine. Note that in most cases if **Repair** was not helpful it also provided little output to waste a user's time: it only once presented more than 3 statements to examine. In practice, we only suggest using **Repair** at all in cases where 1) there is at least one repairing mutant (not killed by any passing test) for a failing test and 2) there are multiple failing tests (whether from one or multiple faults). When the Stmts column shows a bold 0, it indicates there were no repairs (7 of the 30 faults). The 7 faults meeting both requirements are bolded. For 4 of these, **Repair** produces a perfect localization, and for the other three, it produces no more than 2 non-faulty statements to examine.

DOI: 10.1002/stvr

5.5. Discussion

In one sense, the discovery curve improvements here are practically very useful, but not extremely large in a relative sense. The SpiderMonkey curve has an APFD only slightly more than 2% better than the best result from previous FPF efforts. For GCC, APFD improves by 6.4%. However, the comparison is with the very *best* curve chosen after running more than 16 different metrics. In practice, users simply do not know ground truth to rank curves, thus Chen et al. [16] do not really give a practical approach to distance metric selection. The best methods for different subjects varied widely, even in such difficult-to-understand ways as less-fine-grained coverage providing better results in some cases, but worse in other cases. In practice, their work established that FPF could produce good curves, but gave very little useful guidance for choosing a metric for actual use of the technique, since trying multiple metrics is impractical: a user has to examine each curve. The set of metrics also included methods such as comparing output signatures that are inapplicable to differential testing for wrong-code faults, the most critical compiler application. This paper presents a single curve using a universal metric, and achieves a 2-6% percentage point improvement over the best of more than 16 different metrics studied in the previous work; our improvements over any single "reasonable" method applied to both problems would be considerably larger (for instance, using the most obvious basis, line coverage, we see more than 17% improvement). Moreover, the improvement is, for our subject programs, reliable: choosing random subsets of the full data set, the difference in mean APFD from best-previous method is around 3%, and statistically significant.

For fault localization and error explanation, the results show possible improvement on state-of-the-art mutation-based approaches. A practical impact of the results reported is that we suggest users of our techniques examine only the first few (at most 10, and we propose as few as 5) mutants and coverage-changing statements, and ignore localization if none of these results are helpful. This is analogous to our suggestion that a user abandon the FPF curve if 5-10 test cases in a row fail to reveal any new faults, as a heuristic. We suspect the information in the mutants/coverage changes is probably helpful in some cases where they do not localize a faulty line, but this is simply based on our highly incomplete understanding of the faults and patches in question, and not a solidly established claim. The expertise of compiler developers would be needed to confirm or reject this belief. Our suggestion that mutations themselves provide interesting error explanation is also applicable to MUSE and MUSEUM. There may be some potential for confusing explanations, however, if repairs include mutants that also cause some failures, as in the standard MUSE/MUSEUM approach.

For the compiler faults, if any repair for a failure modified a faulty statement, **Repair** always ranked it in the top 34 localizations, ranked it in the top 5 in 6 of 10 such cases, and 3 times ranked it 1st. MUSEUM, in contrast, ranked one such repair 195th, never ranked a fault in its top 3 statements, and only ranked a fault in its top 5 statements twice. Of course, MUSEUM might be suffering from a lack of mutation operators, but in some cases this is unlikely due to the nature of the repairs. For the CoREBench faults, **Repair** produced more than twice as many perfect localizations as the other methods, though it did not provide any ranking for a fault in a large number of cases.

There are two ways to consider the performance of **Repair**. First, it is using information that other techniques are not using. If each failure was only repaired by one mutant, MUSEUM and **Repair** would both rank that mutant maximally, and we speculate that it would usually be faulty code. But such cases are very rare in compilers: for SpiderMonkey the mean/median number of mutants repairing each failure were 90.8 and 78, respectively, and for GCC the mean and median were 54 and 43. **Repair** can distinguish repairing mutants in fine-grained ways that MUSE and MUSEUM cannot by relying on the onion-ring structure: when a repair also repairs failures that otherwise do not resemble the failure being localized (in terms of its repairing mutants), **Repair** assumes that repair is general to many different faults, and so ranks it lower than more "relevant" repairs. MUSEUM relies on a large set of mutation operators, and assumes that faulty statements will have multiple repairs, compared to non-faulty statements, or that faulty statements will be less likely to cause passing tests to fail when mutated. Restricting ourselves to cases where **Repair** and MUSEUM both provided a localization, we know that some faulty statement was in a repairing mutant, but that passing tests could not distinguish this mutant from other repairs (since we only

DOI: 10.1002/stvr

use repairs that cause no passing tests to fail). Many of the faulty statement repairs (as expected for optimizing compilers) were statement deletions and conditional negations. For these, it seems unlikely that Proteum would produce more mutants repairing the statement, though perhaps Proteum would create repairs for other statements that cannot be modified into a repair by any of Andrews' operators. Our suspicion is somewhat confirmed by the results above where we used our methods with the MUSEUM mutant result set, and obtained similar results.

The other way to think about **Repair** for the non-compiler faults especially, is as a very *cautious* localization. It only uses mutants that repair a failure, and, because it is clear that, in general, failing tests contain far more information about faults than passing tests, it only uses information from failures (other than using passing tests to prune mutants that kill some passing tests, again a principle of discarding potentially confusing information). This results in **Repair** either providing a "useful" localization or almost no localization information at all to mislead a user, for the non-compiler faults, which user studies and surveys suggest is the ideal behavior due to the cost of false positives in localization [75, 54]. This is basically a trade-off. MUSE, MUSEUM, and Metallaxis-FL provide localization for many more faults, in a more diverse range of settings (with single failing tests, no repairs, etc.), and using more diverse information—but, in cases where **Repair** provides a localization, it seems to often be higher quality due to its restriction to a high-signal source of information (mutants repairing failing tests, only). In settings, unlike compilers, where there are relatively few repairs, **Repair** also produces very little incorrect information, and so has a low cost to the user in terms of wasted attention. Note that our heuristic to stop reading after 5 wrong localizations is not even needed for all but one CoREBench fault—**Repair** suggests 1-3 statements at most.

Finally, in contrast to many previously reported results in fault localization (which are typically over much simpler programs and faults than subtle compiler semantics bugs) all methods we applied very frequently failed to provide a useful localization at all for our compiler faults. This is presumably due to the sheer difficulty of compiler bugs: our approach, and MUSEUM, performed very well on complex multi-lingual faults in the MUSEUM data set. In one sense, **Coverage** is the most intriguing method presented here. While it did not perform nearly as well as **Repair**, it did manage, in many cases, to rank *some* faulty line highly, relative to the extremely large number of statements executed in each failing GCC run (on the order of 50K-100K+ statements). In future work, we plan to experiment with different methods for weighting frequency of coverage, perhaps using methods from machine learning, or of pruning changed coverage that is not likely to be relevant. For example, perhaps the **Repair** approach can be applied, and coverage changes present in very distant failures can be removed, though the argument is less clear, since a coverage change isn't directly associated with the onion-ring structure, and does not directly repair a set of failures. In fact, a study of the importance of delta-debugging in localization [17] showed that even traditional Spectrum-Based Fault Localization (SBFL) can *sometimes* produce good results on SpiderMonkey faults. Such methods, of course, often perform worse than even **Coverage**; e.g., for our fault #8 (their fault ID 115), **Repair** and **Coverage** produced localizations at rank 3 and 6, respectively, while the best SBFL result was well over rank 32. We suspect that combining multiple methods, using some method to cross-assess confidence, might be required to obtain reliably good compiler bug localizations, and assign good confidence scores to localizations. There is much to be done if automated debugging is ever to make the lives of compiler developers easier in most cases [75], and a human study using real, complex compiler bugs would be ideal, since previous studies of localization effectiveness tend to focus on simple tasks even moderately experienced developers can potentially succeed at. Fixing deep optimizing compiler bugs is a very specialized skill, in contrast to general debugging skills.

6. MUTATION ANALYSIS COSTS AND POTENTIAL OPTIMIZATIONS

Our compiler experiments required a very large computational budget. However, our approach is, by design, cheaper than MUSE or Metallaxis-FL, in that, for a program with up-to-date mutation testing results (just knowing which mutants are killed, not a full matrix), it only requires executing

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutants over failing tests. Running each test case under each mutated version of the compiler is cheap: each execution requires around 0.05 seconds for SpiderMonkey, and 0.12 seconds for GCC, on average; moreover, executions can be done in parallel, most failures do not cover most mutants, and vice versa, so the total number of repair checks needed is far less than the product of mutants and failures, and can be spread over many machines.

Mutation cost reduction techniques are also applicable. E.g., trivial compiler equivalence (TCE) [72] can reject equivalent and redundant mutants, removing almost 30% of mutants for benchmark subjects. Other techniques, such as using mutant schemas to avoid having to compile and store each mutant, are also applicable—almost all of the techniques characterized by Offutt and Untch [71] as *do faster* approaches apply, without any loss of ability to identify or localize faults.

6.1. Hot and Cold Running Mutants

Note that the largest cost in our approach, by far, is compiling the mutants. While running all the GCC tests on the covered, compilable, test-passing mutants takes nearly 24 days of total execution time, the process is embarrasingly parallel: no mutant/test execution depends on any other. On the other hand, all executions of a mutant/test pair depend on having a compiler that includes that mutant built, and compiling GCC requires more time than test executions by a large margin.

We expect that fuzzer taming will mostly be performed when someone is fuzzing a compiler either for the first time or using a new technique. If fuzzing is performed on a well-fuzzed compiler, with only incremental changes from previous versions, there is not likely to be a significant fuzzer taming problem: fuzzing will often only detect a small number of faults, and manual triage efforts will probably suffice. In the case of "new" fuzzing efforts, however, they will usually be aimed at release versions of a compiler. For major projects such as GCC, LLVM, or V8, producing and publishing a set of "hot" mutants might be one solution. *Hot* mutants are mutants that 1) compile 2) pass all current tests and 3) are not known to be equivalent mutants. Producing such mutants should be an easily automatable task without prohibitive cost, at the level of a major open source project. Release versions are only produced a few times a year (e.g., both GCC and LLVM had 5 releases in 2018 and 4 in 2017). In addition to fuzzer taming, such mutants would provide a useful quick benchmark for new compiler-fuzzing techniques: if a technique can kill mutants not detectable by the compiler's test suite, it has possibilities for bug detection as well.

Alternatively, a radical way to produce "hot" mutants at low cost (the only significant cost being that of executing the test suite to throw out mutants killed by some passing test) is to produce mutants at the *object code* level, which does not require recompiling a system. This is how SQLite applies mutation testing: `https://www.sqlite.org/testing.html#mutation_testing`. Object level mutants are obviously not useful (or at least, certainly not *as* useful) for fault localization as source-level mutants, but mutating branches alone probably captures many compiler repairs that are useful in fuzzer taming.

6.2. Lossy Techniques

6.2.1. Random Sampling Mutation cost reduction techniques classified as *do fewer* on the other hand involve a loss of signal, not just more efficient production of the same signal. One of the most popular and simple approaches to reducing the cost of mutation testing is *random sampling* [9, 3, 29, 109]. Figures 9 and 10 show, respectively, the results of random sampling of mutants for SpiderMonkey and GCC, by comparing the APFD obtained on a set of 30 random samples of the chosen percentage of mutants with the APFD for the full mutant set. Because some mutants are actually misleading with respect to distinguishing faults, sometimes sampling can improve results. Sampling 10% of mutants directly translates to our approach requiring about 90% less time (or less computing power if run in parallel). We assume a "hot" mutants approach in that our sampling is of compilable, test-passing mutants. A larger sample would be required, in proportion to how many mutants compile and pass tests, to achieve the same results in a scenario with only "cold" mutants, though the difference is not huge, since most mutants are ruled out by the fact that no failing test covers the mutant, rather than that they cause some passing test to fail. The more significant difference in a cold-mutant scenario is that each mutant/test evaluation is of course much cheaper

DOI: 10.1002/stvr

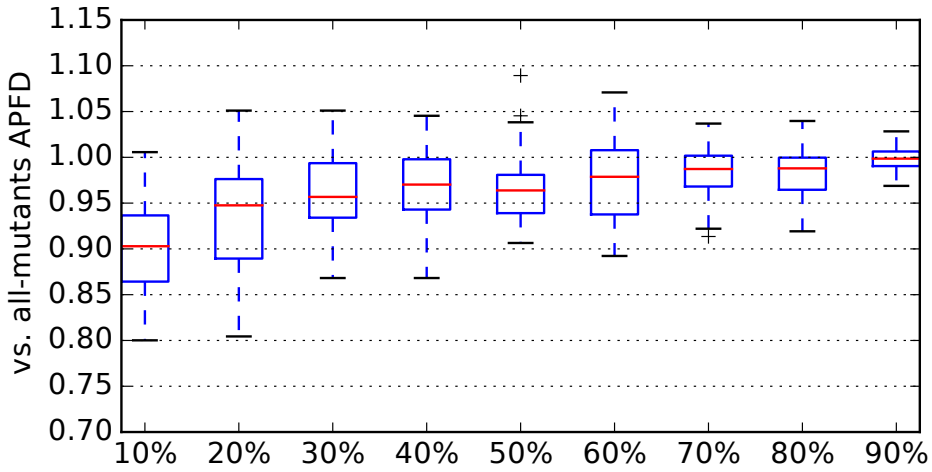

Figure 9. APFD values for Spidermonkey samples

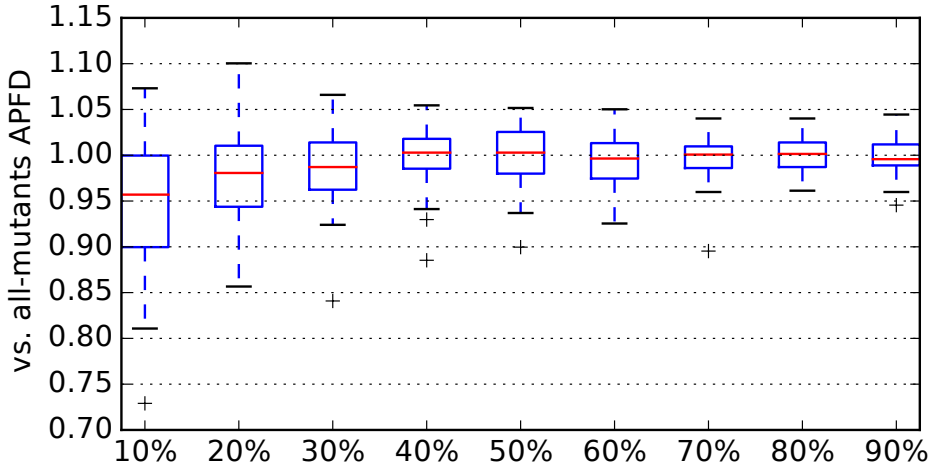

Figure 10. APFD values for GCC samples

without the cost (even amortized over many tests) of compilation, not the total number of mutants considered.

Pure random sampling is fairly effective for GCC, but more mutants are required to obtain a comparable APFD on average, for SpiderMonkey. The reason for this difference is not easy to understand. At a high level, every mutant that repairs at least one failing test test case provides at most one bit of information for fuzzer taming. Some repairing mutants repair all test cases (thus providing no information to distinguish faults), and others provide potentially redundant information in that they repair the same set of mutants as another mutant. Mutants that repair no test cases provide no information, of course. Given that more SpiderMonkey mutants (12.6%) repaired a test case than did GCC mutants (7.8%), it would seem that a *smaller* sample of mutants would be required to produce the "same number of bits of information" with SpiderMonkey. However, SpiderMonkey mutants are more redundant than GCC mutants, and, perhaps more importantly, the SpiderMonkey code is less well-structured, and changes behavior for more causally complex (and unrelated to faults and clear optimization levels) reasons than the more mature GCC code, making each mutant's bit of information inherently more "noisy." For GCC, using only 30% of the mutants

DOI: 10.1002/stvr

produces results quite close to the optimal results, and almost never drops below 90% effectiveness. For SpiderMonkey, similar results come with 60% of mutants. Sampling is an inherently incremental process: mutants can be processed in a random order, and a developer can compute the curve at any point in time. We observed that the early curve is reliably good for both SpiderMonkey and GCC, even with as few as 10% of mutants; most variance is in the further reaches of the curve. If developers can identify and fix bugs based on early mutant results, these can be fed back into the process, and mutants not covered by any remaining unfixed tests can be removed from the set. For fault localization, with hot mutants a single failing test's complete repair set can be found with relatively few evaluations, and then the repairing mutants can be evaluated on all other tests, to compute a localization that is essentially as accurate as the best ones we produced (distances may be slightly off due to mutants repairing other tests, but not the test being localized, but the sampling is likely to largely handle this issue, since exact distances are not important, only relative ones).

We tried various ways of improving the effectiveness of random sampling by forcing the inclusion of certain mutants that were deemed more likely to be useful. For example, it is very important to include mutants that repair hard-to-repair faults represented by a small number of tests. Finding faults represented by few tests is, in some sense, what users want from a fuzzer taming algorithm. We cannot, without evaluating them over tests, predict which mutants will repair only a small number of tests (and thus, likely, faults with only a few representatives). However, making sure to include mutants that are only *covered* by a small number of tests lets us ensure we cover mutants that, if they repair any failing tests, only repair a small number of failing tests. However, due to the likelyhood that such mutants repair no tests at all, including mutants covering at most k tests (for small k) turned out to be either actually harmful (statistically significantly worse APFD) or statistically indistinguishable from pure random selection. We were unable to find any mixes of random and biased or chosen sampling that performed better than pure random sampling. Is this the best we can do?

6.2.2. Operator-Based Selection A deterministic alternative to random sampling, long used in the mutation testing community, is operator-based selection, where only some mutation operators are applied [99, 92]. Using *only statement deletion and condition negation mutants*, as in our toy example, on the grounds that we expect most compiler "repairs" to simply avoid an optimization, not actually fix a simple defect such as an off-by-one error in a loop, is an intuitively appealing and trivial to implement approach. The idea is clearly related to the object-level branch-mutation proposed above, conceptually, but operating at the (more understandable to humans, and thus useful in localization) source level. The resulting cost reduction is significant: 67.8% of mutants can be ignored for SpiderMonkey, and 73% for GCC.

More importantly, the APFD actually *improves* with this change. For SpiderMonkey, the APFD improves by almost 3%, and for GCC it improves by 0.86%. For SpiderMonkey, the early curve (first ten items) is improved, and for GCC it is very slightly worse. Most importantly, in both cases, *two additional faults* are revealed in the first 50 examined tests. We speculate that the accidental nature of many repairs produced by operator and constant replacements may introduce harmful noise into the distance metric, and thus the FPF-based fuzzer taming process. Changes that simply sidestep faulty functionality are, perhaps, more likely to be meaningful [80].

Operator selection can, in principle (and in one case in our data set, in practice) reduce the effectiveness of fault localization. However, with hot mutants, it is easy, if localization using only negations and deletions is not helpful, to use the same method as proposed above for random selection to compute a more precise localization for a test. This should usually not be required, however: we noted that in most cases where a mutant involved in a localization was an operator replacement, the mutant was usually equivalent in behavior to a condition negation. Constant replacement repairs were simply very rare and not involved in any of the useful compiler localizations. The problem observed with random sampling, that the metric of distances to other tests may be inaccurate, is of much less concern with operator selection: as far as we can tell, based on FPF discovery curves and APFD, distances are *more* accurate using only the mutants we propose.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

While using only some mutation operators *may* degrade fault localization performance for non-compiler faults, we propose it as the default way to use our approaches in compiler fuzzer taming and debugging scenarios. Furthermore, we speculate, based on the results of Qi et al. [80] showing that most actual correct fixes in program repair are functionality deletions, that limiting mutants used to condition negations and statement deletions (or even adding an operator that converts conditionals to a `False` constant, as is provided by some tools [36]), might be the best practice for complex programs and "deep" (hard to debug) faults in general. It seems unlikely that mutation operators will produce many fixes that do not simply avoid broken functionality (vs. actually fix it), for the kinds of faults that are actually hard for programmers to locate, understand, and correct.

6.2.3. Combinatorial Control of Optimizations Finally, if we assume that in the setting of compilers, faults are due to optimizations, it may be possible to use a combinatorial approach to turning on/off optimizations as a cheap equivalent of mutants; Ghandehari et. al [87] proposed a combinatorial approach to fault localization and such a method might work fairly well in this setting. Further experimental effort would be required to determine how useful such a method would be; a major concern is that rather than simply "turning off" optimizations as one might at the command line, some mutants change the result of a check as to whether an optimization is valid for a piece of code. GCC and LLVM also do not, unfortunately[‡], expose all the optimizations they perform as individually controllable by users. Based on our examination, the level of control over optimizations provided by GCC 4.3.0 was simply too coarse-grained to enable effective replacement of mutants, and SpiderMonkey 1.6 offers almost no control over optimizations, as would be expected of a JIT usually used in a browser context; it does not work as a command-line tool that takes an input file and produces a binary. There may be API calls to control optimization in some way, but these were not easy to discover, or practical for a developer to use.

7. FUZZER TAMING FOR A NON-COMPILER TARGET: FUZZGOAT

Fuzzgoat (`https://github.com/fuzzstati0n/fuzzgoat`) is a well-known backdoored C program used to test fuzzers and other analysis tools. It is an interesting subject to consider non-compiler fuzzer taming because it is a single program version with 4 distinct bugs with complete ground truth, and widely viewed as representative of a typical fuzzing use-case. We ran AFL [105] for over 7 hours on the program, generating 4,244 crashing inputs (Figure 11 shows a screenshot from just before we terminated the run). AFL's internal triaging methods reduced this to a set of 38 crashes it considered distinct. We terminated the run once it had been nearly three hours since a new distinct crash was detected.

These crashes exhibit a classic fuzzer taming structure. Three of the bugs are easy to identify: they are triggered by 15, 14, and 11 of the 38 AFL-produced crashing inputs, respectively, and all have multiple inputs that uniquely detect that bug, and no other. The fourth bug, however, is only triggered by *three* of the 38 inputs, and is never uniquely present (any input triggering it also triggers at least one other bug). The bug frees a pointer that can subsequently be used, and it is (as often with use-after-frees [93]) hard to produce inputs that actually cause this to happen. This bug was only found by AFL after nearly an hour of fuzzing, in fact.

This is a classic fuzzer taming problem, albeit on a smaller scale than with the compilers: a user will probably not want to inspect the highly-redundant set of 38 inputs, but will want to see all four distinct faults, while examining as few inputs as possible.

Applying the mutation-repair based approach is relatively easy here. We used the universalmutator tool [36] (`https://github.com/agroce/universalmutator`) to generate 18,489 mutants for fuzzgoat. This tool is far more aggressive than Andrews' tool, in terms of operators used, and, more importantly is more widely available to typical users and easy to apply

[‡]Unfortunately, at least, from this combinatorial point of view; as developers we may suspect that modern compilers already have plenty of command-line options.

DOI: 10.1002/stvr

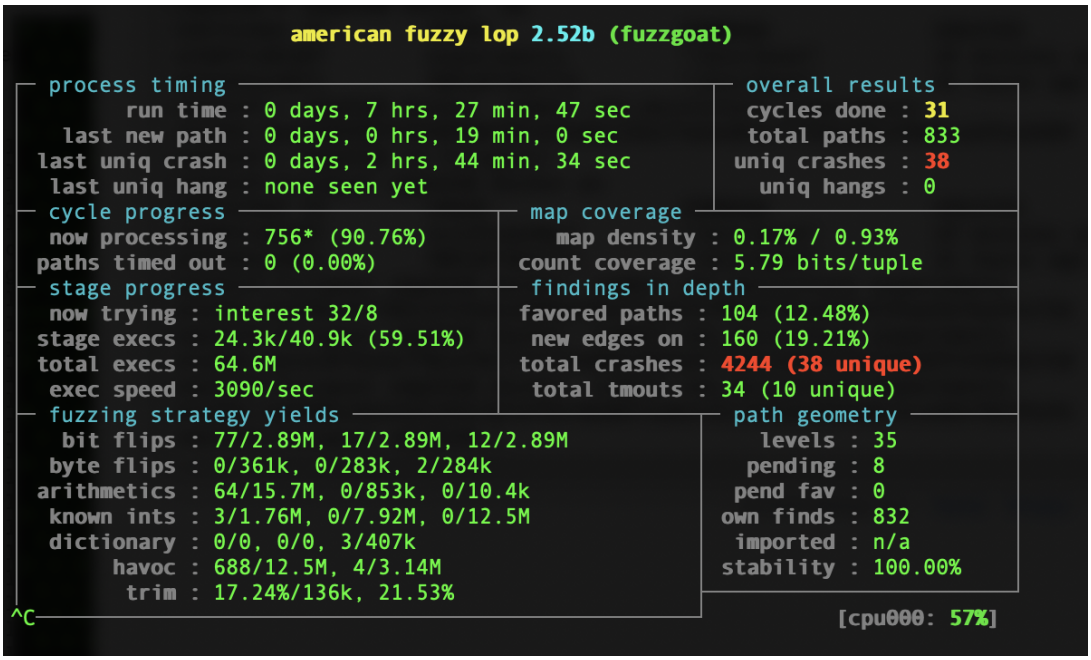

Figure 11. AFL run for fuzzgoat

to source code in a large set of languages, unlike, e.g., Proteum. Of these mutants, only 6,809 actually compile, and trivial compiler equivalance [72] can further reduce the set to 2,899 mutants to be used in producing repair bitvectors. Computing the full bitvector set and all distances for the 38 crashes required just over 10 minutes. While compiling and running mutants is a serious computational burden in the setting of a production compiler, it is negligible compared to the time required for fuzzing, in a typical fuzzing effort, we suspect, even without using such simple additional reductions as checking the mutants for code covered by some crashing input. This is partly because the number of mutants to consider is smaller, and partly because compiling GCC or SpiderMonkey is simply much slower than compiling more typical small-to-medium-sized fuzzing targets. Fuzzing a very complex target, such as a complex production-grade database (e.g., LevelDB or RocksDB) or the Linux kernel itself might be more like compiler fuzzing, however, in terms of mutation analysis cost. Input runtime is also smaller for most non-compiler targets, we suspect, even including "hard" targets like databases and kernels.

Table IV shows fuzzer taming results for fuzzgoat, using our repairing-mutant-based approach, Chen et. al's [16] statement coverage method, and (as a baseline) random ordering. For repair-based ranking and statement coverage-based ranking, we show the mean result over all 38 possible choices for first test to rank. Random is the mean over 38 random rankings. **Cutoff** indicates the point at which the user stops examining crashing inputs, and **Mean bugs** indicates how many of the four distinct faults, on average, the method, with that cutoff, allowed the user to see within that cutoff. **# Perfect** describes how many of the 38 experiments produced a perfect triage, where all of the bugs were seen before the cutoff. We begin **Cutoff** at 3, since 3 is the minimum possible number of inputs a user must examine to find all four bugs (some inputs cause a crash due to either of two bugs, so are able to identify two faults). Coverage-based FPF is better than repair-based FPF, for fuzzer taming, if a user only examines 3 of the 38 crashes. However, once a user is willing to look at 4 inputs, mutation-repair is better, and if the user is willing to look at 5 of the 38 inputs, our approach guarantees all of the distinct bugs will be seen. Statement coverage-based FPF, in contrast, requires a user to look at **18** inputs before providing such a guarantee. Indeed, there is a more than 25% chance of missing one bug even if the user looks at 10 inputs, and random ranking performs almost as well as coverage-based FPF, once a user is willing to examine 10 or more inputs. In the table, cutoffs for which statement-based coverage performance is identical to the previous cutoff are

DOI: 10.1002/stvr

Method	Cutoff	Mean bugs	# Perfect	% Perfect
Repair	3	3.00	0	0.00%
Coverage	3	3.10	13	34.21%
Random	3	2.58	3	7.89%
Repair	4	3.42	16	42.10%
Coverage	4	3.39	15	39.47%
Random	4	2.74	6	15.79%
Repair	5	4.00	38	100.00%
Coverage	5	3.50	19	50.00%
Random	5	3.29	16	42.10%
Repair	6	4.00	38	100.00%
Coverage	6	3.50	19	50.00%
Random	6	3.18	11	28.95%
		...		
Repair	10	4.00	38	100.00%
Coverage	10	3.63	24	63.16%
Random	10	3.58	23	60.53%
Repair	11	4.00	38	100.00%
Coverage	11	3.63	24	63.16%
Random	11	3.66	25	65.79%
		...		
Repair	17	4.00	38	100.00%
Coverage	17	3.97	37	97.37%
Random	17	3.84	32	84.21%
Repair	18	4.00	38	100.00%
Coverage	18	4.00	38	100.00%
Random	18	3.82	31	81.58%

Table IV. Fuzzgoat triage results

omitted. That the coverage-based ranking does not improve its performance between cutoff of 6 and 9, and 11 and 16, shows how weak the signal from the distinct faults is, in terms of code coverage.

To our knowledge, the state-of-the-art in research on advancing fuzzing triage over the tools built into fuzzers like AFL is represented by the *semantic crash bucketing* (SCB) approach of van Tonder, Kotheimer, and Le Goues [94]. This approach uses patch templates based on common types of bugs/semi-fixes, combined with semantic information obtained by observing the execution of a crashing input, to bucket crashing inputs discovered by fuzzers. We contacted the SCB authors, and with their cooperation investigated how well SCB handled the fuzzgoat crashes. The fuzzgoat code highlights the primary limitation of SCB: namely, if a kind of failure is not handled by a (manually constructed) patch template and identified by the semantic feedback mechanisms, then SCB does not really handle it. The SCB tool was able to produce a patch for one of the four fuzzgoat bugs, a null pointer dereference. Unfortunately, at present, SCB does not handle bugs related to misuse of `free` (use-after-free, double frees, or frees of invalid pointers), and the other three fuzzgoat bugs are all `free`-related. SCB therefore can very effectively identify the inputs associated with one of the four bugs (when it is the only one triggeered by the input), but provides no assistance with finding the other three bugs. This is not to say that SCB is useless here; the null pointer bug alone is responsible for 13 of the 38 crashes, so SCB does make it easy to see one example of the null pointer bug, and ignore another 12 inputs. It does not, however, help find the rare bug with only a few instances among the remaining 25 inputs. While mutation repair also has no knowledge of `free` bugs, it is able to find mutants that avoid their symptoms or their causes, and thus triage all four faults effectively. The SCB team added the fuzzgoat example to their published VM, and plan to use it to drive development of patch templates and analysis for use-after-frees; this will, at some point, likely make it possible for SCB to handle fuzzgoat effectively. However, in the long term, if fuzzing is applied to find assertion violations, differential testing divergences [66, 37], or other more complex "custom" properties, as is likely with the appearance of tools such as DeepState [27] that merge fuzzing and property-based testing, it is hard to see how SCB will handle these "custom" bug types.

DOI: 10.1002/stvr

Fuzzgoat is a poor candidate for fault localization, due to the lack of a reasonable set of passing tests, and due to the fact that all four bugs are fairly easy to understand once an appropriate input is run under a sanitizer. We do note that repair manages to localize the null pointer dereference perfectly (SCB's patch is also an excellent localization for this bug). Given passing tests, we suspect all fault localization methods discussed in this paper could handle the localization problem easily, in contrast to a typical compiler bug.

8. THREATS TO VALIDITY

The largest threat to validity is that this paper's core conclusions rely on only two data sets for compilers, and a small number of non-compiler faults. The compiler ground truths are possibly imperfect, due to the complex change history, and it was not possible to produce good patches for all faults, limiting our analysis of compiler fault localization. The comparison with other fault localization methods is also limited: our compiler experiments limit the mutation budget in ways that may weaken MUSE and MUSEUM effectiveness, and the results using fault and mutant data sets from other studies provide few faults, no multi-fault problems, and very few multi-failure problems, limiting **Repair** effectiveness. It is also possible that faults in the framework used for experiments introduced errors into our results. In order to make it possible to check for these problems, we have made the raw data for a re-analysis of the PLDI13 benchmark available, including the tests, ground truth, and mutants, at `https://github.com/agroce/compilermutants`; the fuzzgoat example is also available, including source code for a simple version of our FPF algorithms, at `https://github.com/agroce/fuzzgoattriage`. The data required for analysis of the non-compiler faults is available from the authors of the respective studies (see the acknowledgements at the end of the paper).

9. RELATED WORK

There are four primary areas of related work. First, fuzzer taming and related problems of determining the set of faults from a large set of redundant failures have been investigated in a few papers. Second, some recent efforts to localize faults have begun to incorporate the use of program mutants, and the broader problem of fault localization has been very extensively investigated. Program mutants have also been used in efforts to produce automatic repairs for program faults. Finally, our work fits into the general framework of efforts to define distances between program executions.

The most closely related work (other than the ISSRE 2018 [43] paper that we extend) is that of Chen et. al [16], whose FPF [25] algorithm and benchmark we use. We extend their work with a better, and more universal distance metric, as well as an approach to fault localization in addition to fuzzer taming. Prior to the FPF-based work, Francis et al. [23], Podgurski et al. [78] and others [63, 62] used clustering to attempt to solve similar problems in identifying distinct faults, on simpler human-generated data sets. Chen et. al [16] provide an in-depth comparison of clustering and ranking, and the advantages of ranking (as well as some possible limitations). Jones et al. discuss general issues in debugging multiple faults [49]. The work of Groce et al. [35] provides an alternative approach to fault identification, based on term rewriting to normalize failures, but is not currently applicable to compiler fuzzing, as it only supports tests that are sequences of method calls in Python [39, 44].

One recent approach does, at a conceptual level, resemble our methods: *semantic crash bucketing* (SCB) improves the fuzzer taming ability of AFL [105], Honggfuzz [28], and CERT BFF [89], by approximating real bug fixes with lightweight program modifications [94]. The primary difference between SCB and our approach is that we generate many more potential approximate fixes, via mutation, but use no semantic feedback from failing inputs other than code coverage. The selectivity of SCB dramatically improves its scalability and ease-of-use, but limits applicability to cases where a patch template has been manually constructed, e.g., buffer overruns and null

DOI: 10.1002/stvr

pointer dereferences. While these cases are by far the most common in traditional security/crash-based fuzzing, crashes are usually the *uninteresting* cases in compiler fuzzing, or fuzzing/symbolic-execution based property-based [18] testing [27, 26, 91]. It is unlikely that enough patch templates for the interesting changes to compiler code, or invariant-preserving code in other fuzzing, can be manually produced to make SCB alone generalize to these settings. Another difference is that SCB provides bucketing, rather than ranking; this is in some cases ideal (a user of AFL usually just wants to count "different" vulnerabilities that are memory-safety based), but in settings where the fixes are even more approximate and overlapping, and the bugs more complex, ranking is most useful.

Fault localization [98, 82, 61, 62, 19, 51, 50, 49, 32, 11, 63, 31, 85, 2, 102, 107], even specifically for compilers [97], is a long-standing field of research. Recently, fault localization researchers have used program mutants to improve statistical [51] fault localization techniques [68, 20, 24, 45, 74, 12]. The core of this work has been the belief that faulty program locations are (1) more likely to change a failing test to successful when mutated and (2) less likely to change a successful test case to faulty when mutated. This clearly connects to our general notion that how two test cases respond to a set of mutants is a good measure of their semantic similarity. The work of Hong et al. [45] on MUSEUM and Moon et al. [68] on MUSE provides a good summary of other work along these lines, and is perhaps most similar to ours in assumptions. In particular we agree with "Conjecture 1" that a test case that used to fail on a program is more likely to pass on a mutation of a faulty statement. However, while we suggest some localizations as a welcome and interesting side effect of our distance metric and FPF analysis, our primary goal is the discovery of all faults. Much fault localization work is largely (in many cases exclusively) based on single-fault scenarios; e.g., the MUSE [68] evaluation includes only 2 of 14 scenarios with multiple faults, and those scenarios include only a few faults. The statistical approaches may owe some of their superiority in experiments over other methods to the use of mostly single-fault evaluations [50]. By focusing on a single failing test case (for us, selected by fuzzer taming) as the locus of analysis, our work, like some early efforts [82, 19, 32] may be more suited for situations involving many faults. MUSEUM [45] also revisits this approach, combining the MUSE formula with localization using only a single failure. A recent survey of advances in mutation testing effectively summarizes much of the recent work on using mutants in fault localization [73].

Explaining, rather than merely localizing, bugs is a less common goal, though all mutation-based approaches can probably be adapted to this idea (since a mutation is, itself, a kind of explanation). Key work on explaining bugs in addition to localizing them includes Groce's early model-checking work [32, 11] and the MintHint [53] approach, which bears some resemblance to the use of mutants as "hints" for fault repair, but is more general and backed by user studies to show its utility. The real-world value of MintHint explanations supports the idea that mutant-based explanations might be useful to developers.

Using program modifications or mutants broadly defined to repair programs has been a popular topic of recent work as well [57, 79, 96]. These *generate-and-validate* approaches to patching aim to fix programs by generating a large variety of potential fixes, then using test cases to prune out invalid fixes. Qi et al. [80] discuss some serious limitations of early work on this topic, but also demonstrate that effective patching is possible using a restricted search space that focuses on removing functionality.

Some approaches to localization or patching assume the possibility of a mutant/change *actually* fixing a fault. In our experiments, we found this to be highly unlikely – most of our compiler patches were far larger than the largest higher-order-mutants [47] anyone usually considers applying, and *none* were equivalent to a repair. Previous work has shown that, across a large body of open source programs in many languages, most fault fixes were larger in size than even low-order higher-order mutants [30]. We speculate that fixes for compilers may be likely to be even more complex than the average, given that compilers are particularly complex software systems. Our primary assumption is simply that similar executions respond similarly to mutants, and in particular executions due to the same fault can be "repaired" by similar program changes. These changes are likely to be *related* in some causal way to the actual fault, but are unlikely to be the actual fault, for reasons of comparative complexity of mutants and real fixes.

DOI: 10.1002/stvr

The use of distance metrics in software engineering includes some of the localization and fault identification efforts discussed above (e.g., [82, 32, 11, 63]). Vangala et al. proposed using distance to cluster test cases to improve diversity and eliminate duplicates [95]. We speculate that using FPF with our causal metric over passing tests as a test prioritization (or selection) method might be a fruitful approach [104]. Adaptive random testing uses distances (usually between inputs) to choose tests [14, 15, 5], and Artzi et al. guided test generation for fault localization using path constraint metrics [6]. Sumner et al. evaluated approaches to generating execution peers for debugging and understanding [90] using tree edit distances (and evaluated other distances). Xin et al. [101] examined methods for indexing program executions, useful in all methods involving execution alignments. To our knowledge, most metrics used are essentially spectrum-based [83, 7] (using counts over structural entities), while our metric is essentially causal, based on mutants as counterfactual [59, 60] versions of a program.

10. CONCLUSIONS AND FUTURE WORK

When are two failing test cases due to the same fault? This paper demonstrates that "when they are repaired by the same program mutants" is a useful answer to this question. The problem of causality in executions is hard, and has long been tied to a notion of similarity of executions [82, 32]. By using program changes (produced by mutants) as causes to determine distance, it is possible to improve on efforts to identify the set of faults in a large set of redundant compiler failures [16] and discover the source code locations of faults [68, 45]. For a set of more than 2,800 tests and more than 50 faults, over the Mozilla SpiderMonkey 1.6 JavaScript compiler and GCC 4.3.0, our methods improve fault identification ("fuzzer taming"), in a statistically significant way (with effect size of about 3% improvement), over the *best* (for each subject program) of over 16 metrics considered in previous work.

We also demonstrated that our approach can be cheap and effective in fuzzer taming not targeting compilers. It performed much better than a coverage-based ranking approach, reliably revealing distinct faults in the first few inputs presented, and is more general in application than the (to our knowledge) best competing method, semantic crash bucketing [94].

Preliminary investigation also shows improvement (for a shared mutation testing budget) over previous mutant-based fault localization methods, for the same tests and a subset of 24 faults with known locations. Our **Repair** method, in particular, was the only localization approach to provide any perfect localizations of faults (it produced 3), and provided three times as many very-high-quality (ranking the fault in top 5 locations) localizations as the next best method [45] — 6 such localizations, compared to only 2. For GCC wrong-code faults, only our methods produced any very-high-quality fault localizations. When applied to non-compiler faults, the metric tended to either not produce a localization, produce a very small incorrect localization (1-3 statements), or produce a very high quality localization. Restricting to the cases where we consider **Repair** applicable (7 faults) **Repair** perfectly localized 4 of them, and produced a 1 or 2 statement incorrect localization for the others.

As future work, we plan to apply the mutation-based metric to other fuzzer taming and fault localization problems. The highly varying results for even the best fault localization methods in our experiments demonstrate that further advances are required before compiler debugging can consistently be made less onerous through automated assistance.

More practically, we would like to tightly integrate our approach with automated test generation tools, such as TSTL [44, 38, 39], DeepState [27, 26], Echidna [88], and Manticore [69]. Automated test generation tools tend to produce a large number of passing and failing tests, and are often used in "overnight" runs where adding a step to use mutants to triage failing tests and provide localization suggestions for the highly ranked tests may not even impose a noticable overhead on current usage practices. The latter two tools (Echidna and Manticore), which target smart contracts for the Ethereum blockchain [100], are particularly promising opportunities, because the gas-limited computation bounds of the Ethereum Virtual Machine essentially guarantee that smart contract code is small enough to make mutation analysis cheap. The availability of a multi-language mutation

DOI: 10.1002/stvr

analysis tool [36] supporting Python, C/C++, and Solidity (the language used for most smart contracts) means that a common infrastructure base can be shared by all of these tool integrations.

We also plan to explore other applications of mutant-based metrics. Many software engineering techniques rely on measuring distance between program executions [7, 82]. Not all of these concern only failing executions. For example, Zhang et al. [108] show that FPF based on a simple Hamming distance over branches covered can significantly improve the effectiveness of seeded symbolic execution [65, 77, 48]. Mutation response is likely too expensive to use for this purpose without modification, but sampling mutants to refine distance when coarser metrics plateau may be practical. Cost may be less of a concern when applying our metrics as a method for prioritizing or selecting regression tests, where the results can be repeatedly used in test suite execution, without recomputing distances based on every code change [104]. Our metrics may also be useful in determining execution peers [90], for example to help operators of spacecraft find similar behaviors to telemetry downlinks in testbed history [41, 34].

Acknowledgements: The authors would like to thank the authors of the MUSEUM [45] and Metallaxis-FL vs. MUSE [12] studies for access to their data sets. In particular, we thank Shin Hong and Thierry Chekam for prompt assistance beyond the call of duty. We would like to thank the authors of the Semantic Crash Bucketing [94] study, in particular Rijnard van Tonder and Claire Le Goues, for their assistance in comparing our results with SCB.

REFERENCES

1. LLVM bugzilla search. `https://llvm.org/bugs/buglist.cgi?resolution=---&query_format=advanced&bug_status=ASSIGNED&product=clang`.
2. Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, pages 39 –46, 2006.
3. Allen Troy Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.
4. James H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411, 2005.
5. Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness. In *International Symposium on Software Testing and Analysis*, pages 265–275, 2011.
6. Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 49–60, 2010.
7. Thomas Ball. The concept of dynamic analysis. In *Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 216–234, 1999.
8. Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 105–115, New York, NY, USA, 2014. ACM.
9. Timothy Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.
10. Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. pages 220–233. ACM, 1980.
11. Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, pages 73–82, 2004.
12. Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Assessing and comparing mutation-based fault localization techniques. *CoRR*, abs/1607.05512, 2016.
13. M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604, 2002.
14. Tsong Yueh Chen. Fundamentals of test case selection: diversity, diversity, diversity. In *International Conference on Software Engineering and Data Mining*, pages 723–724, 2010.
15. Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science*, pages 320–329, 2004.
16. Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Programming Language Design and Implementation*, pages 197–208, 2013.
17. Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *IEEE International Workshop on Debugging and Repair*, pages 184–191, 2018.
18. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
19. Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, pages 342–351, 2005.
20. Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45 – 60, 2014.
21. Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

DOI: 10.1002/stvr

22. Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000.
23. Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *International Symposium on Software Reliability Engineering*, pages 451–462, 2004.
24. Pei Gong, Ruilian Zhao, and Zheng Li. Faster mutation-based fault localization with a novel mutation execution strategy. In *Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, 2015.
25. Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.
26. Peter Goodman, Gustavo Greico, and Alex Groce. Tutorial: DeepState: Bringing vulnerability detection tools into the development cycle. In *IEEE Cybersecurity Development Conference (SECDEV)*, 2018.
27. Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.
28. Google. honggfuzz. `https://github.com/google/honggfuzz`.
29. Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *International Symposium on Software Reliability Engineering*, 2015.
30. Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *International Symposium on Software Reliability Engineering*, pages 189–200, 2014.
31. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
32. Alex Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.
33. Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *International Conference on Software Testing, Verification and Validation*, pages 243–252, 2014.
34. Alex Groce, Klaus Havelund, and Margaret Smith. From scripts to specifications: The evolution of a flight software testing effort. In *International Conference on Software Engineering*, pages 129–138, 2010.
35. Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2017.
36. Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering: Companion Proceeedings*, pages 25–28, 2018.
37. Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.
38. Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.
39. Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.
40. Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
41. Klaus Havelund and Rajeev Joshi. Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In *International Conference on Formal Engineering Methods*, pages 187–202, 2014.
42. Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.
43. Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *International Symposium on Software Reliability Engineering*, pages 166–177, 2014.
44. Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.
45. Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 464–475, 2015.
46. D. Hume. *A Treatise of Human Nature*. London, 1739.
47. Yue Jia and Mark Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379–1393, October 2009.
48. Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *Int. Conf. on Software Engineering*, pages 474–484, 2012.
49. James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *International Symposium on Software Testing and Analysis*, pages 16–26, 2007.
50. James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.
51. James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, pages 467–477, 2002.
52. René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 654–665, 2014.
53. Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. MintHint: automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.
54. Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 165–176, New York, NY, USA, 2016. ACM.

DOI: 10.1002/stvr

55. Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Programming Language Design and Implementation*, page 25, 2014.
56. Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 386–399, 2015.
57. Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.
58. Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
59. D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.
60. David Lewis. *Counterfactuals*. Harvard University Press, 1973, revised 1986.
61. Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language and Design*, pages 141–154, June 2003.
62. Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language and Design*, pages 15–26, June 2005.
63. Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Symposium on the Foundations of Software Enginering*, pages 46–56, November 2006.
64. José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Mutation testing for the new century. chapter Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation, pages 113–116. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
65. Paul Dan Marinescu and Cristian Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.
66. William McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
67. James Mickens. The night watch. *;login: the USENIX magazine*, pages 5–7, November 2013.
68. Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014.
69. Mark Mossberg. Manticore: Symbolic execution for humans. `https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/`, April 2017.
70. Seokhyeon Mun, Yunho Kim, and Moonzoo Kim. Fiesta: Effective fault localization to mitigate the negative effect of coincidentally correct tests. Technical Report CS-TR-2014-386, CS Dept. KAIST, South Korea, February 2014.
71. A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.
72. Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering*, 2015.
73. Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.
74. Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test., Verif. Reliab.*, 25(5-7):605–628, 2015.
75. Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.
76. Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *International Conference on Machine Learning*, pages 727–734, 2000.
77. Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Programming Language Design and Implementation*, pages 504–515, 2011.
78. Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *International Conference on Software Engineering*, pages 465–475, 2003.
79. Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering*, pages 254–265, 2014.
80. Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36, 2015.
81. John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*, pages 335–346, 2012.
82. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.
83. Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 432–449, 1997.
84. Jesse Ruderman. Introducing jsfunfuzz, 2007. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`.
85. Raul Andres Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *International Conference on Software Engineering*, pages 56–66, 2009.
86. David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.
87. L. Sh. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung, and T. Xie. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

DOI: 10.1002/stvr

88. J P Smith. Echidna, a smart fuzzer for ethereum. https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/.
89. Software Engineering Institute. CERT BFF. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=507974.
90. William N. Sumner, Tao Bao, and Xiangyu Zhang. Selecting peers for execution comparison. In *International Symposium on Software Testing and Analysis*, pages 309–319, 2011.
91. Trail of Bits. DeepState: A unit test-like interface for fuzzing and symbolic execution. https://github.com/trailofbits/deepstate, 2018.
92. Roland H Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.
93. Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 405–419, New York, NY, USA, 2017. ACM.
94. Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 612–622, New York, NY, USA, 2018. ACM.
95. Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *Foundations of Software Engineering (FSE/ESEC)*, pages 293–294, August 2009.
96. Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.
97. David B. Whalley. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16(5):1648–1659, September 1994.
98. W. Eric Wong and Vidroha Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, The University of Texas at Dallas, November 2009.
99. Weichen Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.
100. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
101. Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Programming Language Design and Implementation*, pages 238–248, 2008.
102. Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, 2014.
103. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294, 2011.
104. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.
105. Michal Zalewski. american fuzzy lop (2.35b). http://lcamtuf.coredump.cx/afl/. Accessed December 20, 2016.
106. Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.
107. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.
108. Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.
109. Lu Zhang, Shan-Shan Hou, Jun-Jun Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *International Conference on Software Engineering*, pages 435–444, 2010.

DOI: 10.1002/stvr