Registered Report: First, Fuzz the Mutants

Alex Groce
and Goutamkumar Tulajappa Kalburgi
Northern Arizona University
alex.groce@nau.edu, gk325@nau.edu

Claire Le Goues
and Kush Jain
Carnegie Mellon University
clegoues@cs.cmu.edu, kdjain@andrew.cmu.edu

Rahul Gopinath
CISPA, Saarland University
rahul@gopinath.org

***Abstract*—Most fuzzing efforts, very understandably, focus on fuzzing the program in which bugs are to be found. However, in this paper we propose that fuzzing programs "near" the System Under Test (SUT) can in fact improve the effectivness of fuzzing, even if it means less time is spent fuzzing the actual target system. In particular, we claim that fault detection and code coverage can be improved by splitting fuzzing resources between the SUT and *mutants* of the SUT. Spending half of a fuzzing budget fuzzing mutants, and then using the seeds generated to fuzz the SUT can allow a fuzzer to explore more behaviors than spending the entire fuzzing budget on the SUT. The approach works because fuzzing most mutants is "almost" fuzzing the SUT, but may change behavior in ways that allow a fuzzer to reach deeper program behaviors. Our preliminary results show that fuzzing mutants is trivial to implement, and provides clear, statistically significant, benefits in terms of fault detection for a non-trivial benchmark program; these benefits are robust to a variety of detailed choices as to how to make use of mutants in fuzzing. The proposed approach has two additional important advantages: first, it is fuzzer-agnostic, applicable to any corpus-based fuzzer without requiring modification of the fuzzer; second, the fuzzing of mutants, in addition to aiding fuzzing the SUT, also gives developers insight into the mutation score of a fuzzing harness, which may help guide improvements to a project's fuzzing approach.**

I. Introduction

Fuzzing is an essential tool for ensuring that software is robust, secure, and as error-free as possible [**?**]. However, even relatively simple program patterns can cause problems for fuzzing, despite the vast effort devoted to improving fuzzing techniques in both academic and industrial settings, recently.

For instance, consider the problem of fuzzing a program whose structure is as follows:

```
if (!hard1(input)) {
    return 0;
}
if (!hard2(input)) {
    return 0;
}
crash();
```

Assume that conditions `hard1` and `hard2` are independent constraints on an input, both of which are difficult to achieve. A normal mutation-based fuzzer such as AFL or libFuzzer attempting to reach the call to `crash` will generally first have to construct an input satsifying `hard1` and then, while preserving `hard1`, modify that input until it also satisfies `hard2`. A key point to note is that if the fuzzer accidentally produces an input that is a good start on satisfying `hard2`, or even completely satisfies `hard2`, *before "solving"* `hard1`, such an input will be discarded, because execution never reaches the implementation of `hard2` unless `hard1` has already been "solved." There is no reason for the fuzzer's interestingness function to consider such inputs for adding to the fuzzing queue.

Even though the fuzzer must eventually satisfy *both* conditions, it can only work on them in the execution order. By analogy, consider the problem of rolling a pair of *ordered* dice. If the goal is to roll two values above five, and you are allowed to "save" a good roll of the first of the two dice and use it in future attempts, the problem is easier than if the dice have to be rolled from scratch each time (i.e., coverage-driven mutation-based fuzzing is usually more effective than pure random testing). However, it is not as easy as if good rolls of the second die can also be saved, even if the first die has never produced a five or six! In fact, if our die has 1,000 sides, and we want each die rolled to have a value of 998 or above, allowing the second die to be saved reduces the number of required trials by close to one third, and the improvement increases with the difficulty of the second condition[1].

If we fuzz a variant of our example program that is modified to omit the first `return` statement:

```
if (!hard1(input)) {
  /* return 0; */
}
if (!hard2(input)) {
    return 0;
}
crash();
```

then progress towards both `hard1` and `hard2` can be made *at the same time*, independently, in any order. If a generated input progresses achievement of either `hard1` or `hard2` it will be kept and used in further fuzzing. Of course, *crashing inputs* for this modified program are seldom crashing inputs for the original program. However, given a partial or total solution to `hard1` and a partial or total solution to `hard2`, it should be much easier for a fuzzer to construct a crashing input for

International Fuzzing Workshop (FUZZING) 2022
27 February 2022, Virtual
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23xxx
www.ndss-symposium.org

[1]The basic power of coverage-guided fuzzing of course, is even more critical: allowing saving the first die improves the number of rolls needed by orders of magnitude, not a mere large percentage.

the original program. This is a very simple example of a case where fuzzing a similar program can produce inputs such that 1) they help fuzz the actual program under test and 2) those inputs are much harder, or potentially almost impossible, to generate by fuzzing the actual target program.

Three points are important to note about this approach: first, fuzzing an arbitrary program would be of no use here. Inputs useful in exploring that program would likely be useless in exploring the real target of fuzzing. Second, if a modification has little semantic impact on the original program, then fuzzing that variation is, to a large extent, the same as fuzzing the original program, with the only cost being some additional fuzzer logistics overhead. For instance, fuzzing this variation of our example program:

```
if (!hard1(input)) {
    return 1;
}
if (!hard2(input)) {
    return 0;
}
crash();
```

is, for purposes of input generation, no different than fuzzing the original target program. Only exit codes from the program are affected, if the `return` statement appears in `main`. Similarly, a version removing the call to `crash` will still result in the fuzzer attempting to "push through" `hard1`, even though the result of complete success will be less dramatic until the input is applied to the actual target program, and fuzzing

```
if (!hard1(input)) {
    return 0;
}
if (hard2(input)) {
    return 0;
}
crash();
```

is, until `hard1` is covered, indistinguishable from fuzzing the original target program. Of course, not all variants are helpful or harmless. Fuzzing degenerate versions like this:

```
if (1) {
    return 0;
}
if (!hard2(input)) {
    return 0;
}
crash();
```

is obviously a waste of time.

Removing difficult checks is not the only potential win when fuzzing variants. Consider the problem of fuzzing a compiler that includes a very expensive optimization pass. Transforming the code by removing a call to that pass may not make it easier to hit a deep bug in another part of the code, in terms of behavior of the inputs, but might improve fuzzing throughput so much that paths through other parts of the code are explored much sooner, and thus the bug is found much more quickly. In particular, if an optimization pass is very likely to have quadratic or worse behavior on fuzzer-generated inputs, disabling it may be tremendously productive.

Predicting which program variants will aid fuzzing seems inherently hard. In the example, removing a well-chosen statement was extremely useful; in other cases breaking out of a loop before it fails a check (by adding a `break`) or skipping a check in loop (by adding a `continue`) might be important, or turning a condition into a constant true — or constant false! Analysis capable of detecting reliably "good" changes seems likely to be fundamentally about as hard as fuzzing itself, or symbolic execution. Recall however, that many variants that are not useful will also be harmless, in that they amount to simply fuzzing the target. What we need is a source of similar programs that will include the (perhaps rare) high-value variants (such as removing the `return` above), and will not include too many programs so dis-similar in semantics they provide no value.

Program *mutants* provide such variants, by design [**?**]. Mutants are designed to show weaknesses in a testing effort, by showing the ability of a test suite to detect *plausible bugs*. The majority of such hypothetical bugs must be semantically similar enough to the original program that a test suite's effectiveness is meaningful for the mutated program. Therefore, most program mutants will satisfy the condition of being close enough to the target of fuzzing. Mutants are roughly evenly distributed over a program's source code, and modify only a single location. Therefore most uninteresting mutants will generally be harmless, since fuzzing the mutant will be essentially fuzzing the original program, except for a small fraction of code paths. Finally, mutation operators are varied enough to provide a good source of potentially useful mutants. Most importantly, almost all mutation tools include at least statement deletion (to remove checks that impede fuzzing) and conditional changes (negation, and replacement with constant false and constant true). These are the variants with the most obvious potential for helping a fuzzer explore beyond a hard input constraint, as in the example above. All of the versions of the example program shown above, or mentioned in the list of potential ways around `hard1`, are mutants generated by an actual mutation testing tool [**?**].

Additionally, fuzzing program mutants is a *useful activity in itself*. Mutation testing is increasingly being applied in the real-world. A program worth fuzzing is probably a program worth examining from the perspective of mutation testing. Examining mutants not detected by fuzzing can reveal opportunities to improve a fuzzing effort, either by helping it reach hard-to-cover paths or, more frequently, by improving the oracle (e.g., adding assertions about invariants a mutant causes to be violated, or even creating a new end-to-end fuzzing harness when a fault is not exposed by fuzzing only isolated components of a program). Mutation testing of the Bitcoin Core implementation (see the report (https://agroce.github.io/bitcoin_report.pdf) referenced in the Bitcoin Core fuzzing documentation (https://github.com/bitcoin/bitcoin/blob/master/doc/fuzzing.md) revealed just such limits to the fuzzing, despite its extremely high coverage and overall quality. Mutation testing is supported by widely used and well-supported tools, and available for all commonly used (and many uncommonly used) programming languages.

Moreover, by operating at the level of source modifica-

tions to the program being fuzzed, the proposed technique is agnostic as to the actual fuzzer used. There is no need to implement fuzzing-of-mutants for each fuzzer of interest; the method operates completely at the level of orchestration of fuzzing results.

II. Fuzzing the Mutants, in Detail

A. Mutation Testing

Mutation testing [**?**], [**?**], [**?**] is an approach to evaluating and improving tests. Mutation testing introduces small syntactic changes into a program, under the assumption that if the original program was correct, then a program with slightly different semantics will be incorrect, and should be detected by effective tests. Mutation testing is used in software engineering research, occasionally in industry at-scale, and in some critical open-source work [**?**], [**?**], [**?**].

A mutation testing approach is defined by a set of mutation operators. Such operators vary widely in the literature, though a few, such as deleting a small portion of code (such as a statement), negating a conditonal, or replacing arithmetic and relational operations (e.g., changing + to − or == to <=), are very widely used.

In principle, the ways in which mutants could be incorporated into a fuzzing process are almost unlimited. However, the basic approach can be simplified by considering the fuzzing of mutants as a preparatory stage for fuzzing the target, as in the introductory example. The simplest such approach is to split a given fixed time-budget for fuzzing into two equal parts. First, fuzz the mutants. Then, collect an input corpus from that fuzzing, and fuzz the target program as usual, but for half of the desired overall time.

B. Fuzzing: Two Key Decisions

Given a set of all mutants of a target program, and a decision to split a given fuzzing budget into a mutant-fuzzing stage followed by a target-fuzzing stage, there are two major decisions to be made: how to select a subset of mutants, and how to carry out fuzzing the chosen mutants.

1) Choosing the Mutants: First, there needs to be some source of mutants. For generating mutants, we use the regular-expression-based Universal Mutator [**?**] (https://github.com/agroce/universalmutator), which provides a wide variety of source-level mutants for almost any widely used programming language, and has been used extensively to mutate C, C++, Java, Python, and Solidity code. The latest release of the Universal Mutator is also able to use the Comby [**?**] tool (https://github.com/comby-tools/comby) to generate some mutants hard to express as regular expressions, and to prune mutants that are certain to be invalid more efficiently. Any mutation testing tool should work, in principle, although if the fuzzer requires the program to be compiled with special instrumentation, then it is necessary to use source code mutants, rather than bytecode or binary/LLVM bitcode mutants.

For most programs, reasonable (e.g., 24 hour) fuzzing budgets, and approaches to fuzzing mutants discussed below, it is not possible to fuzz all the mutants of the target program. For instance, if a program has a mere 1,000 lines of code, and 2,000 mutants (not an implausible number), a 12 hour mutant fuzzing budget where each mutant is fuzzed for five minutes only allows fuzzing of 144 mutants, less than 1% of the total mutants. Two obvious options offer themselves: the first of these is purely random selection of mutants, under the assumption that we have no simple way to predict the good mutants, and that good mutants will often be redundant. For the second point, consider the example from the introduction. While less effective than removing the `return` statement, negating the condition, changing it to a constant false, or modifying a constant return value inside `hard1` may all allow progress to be made on `hard2` without first satisfying `hard1`. Other changes might relax the most difficult aspects of `hard1` allowing progress on the easier aspects of the condition, and thus progress on `hard2`. Alternatively, even if we cannot predict the best mutants, it might be reasonable to try to diversify the mutants selected using some kind of prioritization. In particular recent work on using mutants to evaluate static analysis tools [**?**] proposed a scheme for ordering mutants for humans to examine, implemented in the Universal Mutator.

The mutant prioritization uses Gonzalez' Furthest-Point-First [**?**] (FPF) algorithm to *rank* mutants, as earlier work had used it to rank test cases for identifying faults [**?**]. An FPF ranking requires a distance metric d, and ranks items so that dissimilar ones appear earlier. 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' force selection of *some* max-min-distance s.

The Universal Mutator [**?**] tool's FPF metric d is the sum of a set of measurements. First, it adds a similarity ratio based on Levenshtein distance [**?**] for (1) the *changes* (Levenshtein edits) from the original source code elements to the two mutants, (2) the two original source code elements changed (in general, lines), and (3) the actual output mutant code. These are weighted with multipliers of 5.0, 0.1, and 0.1, respectively; the type of change (mutation operator, roughly) dominates this part of the distance, because it best describes "what the mutant did"; however, because many mutants will have the same change (e.g., changing + to −, the other values decide many cases. The metric also incorporates the distance in the source code between the locations of two mutants. If the mutants are to different files, this adds 0.5; it also adds 0.25 times the number of source lines separating the two mutants if they are in the same file, divided by 10, but caps the amount added at 0.25. The full metric, therefore is:

$$5.0 \times r(edit_1, edit_2) + 0.1 \times r(source_1, source_2) +$$
$$0.1 \times r(mutant_1, mutant_2) + 0.5 \times not\_same\_file +$$
$$max(0.25, \frac{line\_dist(mutant_1, mutant_2)}{10})$$

Where r is a Levenshtein-based string similarity ratio, $line\_dist$ is the distance in a source file between two locations, in lines (zero if the locations are in different files), and not_same_file is 0/1.

The effectiveness of prioritization is an open question; for the problem of determining mutation score, it is known that mutation selection strategies can sometimes be actively harmful, less effective than purely random selection [**?**]. However, the statistical properties that make purely random selection attractive in predicting mutation score are not as important for using mutants to aid fuzzing.

a) Alternative Prioritizations.: The above prioritization scheme has the appeal that it is computable given only the source code and mutants, and requires no deeper program analysis, dynamic information, or integration with a particular fuzzer. However, an obvious alternative is to prioritize mutants according to their proximity to the *coverage frontier* of an ongoing fuzzing effort. That is, mutants that change code near (in the program-dependence-graph or some other structural representation) *executed branches where both sides have not been taken* would be given higher priority. Mutants of code that is well-covered, on the other hand, or, alternatively, mutants of code that is deep within completely-uncovered code, would be lowered in priority. If we imagine the example proposed in the introduction to include a large amount of additional code, it is easy to see that this would likely prioritize the mutation of the `return` statement after `hard1`.

There are some drawbacks to this approach, however. First, the prioritization may not be as obviously good as it seems at first. Imagine that the `hard1` condition is indeed on the coverage frontier, but that a large amount of additional easy-to-cover but branch-heavy code is present after the `hard1` branch is taken but before the `return 0` statement. The `return` statement will be a low-priority mutant, since it is not at all close to the coverage frontier! Negating the `hard1` condition, of course, may also be helpful, but will not have the very useful feature of allowing progress on `hard1` and `hard2` at the same time. Furthermore, this approach requires previous fuzzing data, and in particular the computation of the coverage frontier.

Other prioritizations are also possible; for example, if we have existing mutation testing results, it may be that mutants that have been killed are more useful in fuzzing, since they clearly produce a semantic change. Equivalent mutants are harmless, but also useless.

b) Full Mutant Analysis, Continuous Mutant Analysis:: Finally, for especially critical fuzzing targets, especially those that are continuously fuzzed in systems such as Google's OSS-Fuzz (https://github.com/google/oss-fuzz), it may be feasible to spend the resources to fuzz *all* program mutants, both in order to identify undetected mutants and to collect the full corpus of inputs generated using mutants. In fact, a CI-style continuous fuzzing effort could in principle alternative fuzzing the target program with a rolling sequence of mutants (ro at least those that ever generated useful inputs), in practice elminating the clear demarction between fuzzing mutants and fuzzing the target.

Finally, while we do not consider the problem here, in repeated efforts it might be useful to reject some mutants as useless based on past results. E.g., if a mutant causes the program to always crash almost immediately, and so a fuzzer generates many crashes (with only one signature) but few or no differing program paths, then the mutant is almost certainly not worth fuzzing again.

2) Using the Mutants: The second key choice is how to *use* the chosen mutants. Assuming a fixed fuzzing budget per mutant, the most basic choice is whether to fuzz each mutant "from scratch" (possibly using any existing corpus for fuzzing the target), which we call non-cumulative/parallel fuzzing, or to use each mutant's output corpus to seed the next mutant, which we call cumulative/sequential fuzzing. The cumulative/sequential approach has two potential advantages:

1) Many mutants that are fuzzed will potentially benefit from the already-fuzzed mutants, so hitting a key location that has been mutated may be more likely; this is based on the same argument as used to support the approach in general.
2) The final corpus from the last-fuzzed mutant will contain few redundancies, reducing processing or fuzzer startup time for the actual target.

On the other hand, cumulative fuzzing forces processing of the corpus after each mutant to remove inputs causing the next mutant to crash, and, more importantly, prevents fuzzing mutants in parallel. The processing cost is due to the fact that before fuzzing a mutant or the target, any input corpus needs, for the AFL fuzzer at least, to be pruned, removing any crashing inputs that did not crash the previous mutant[2] Removing these sequentially, rather than in a single batch after all mutants, may remove inputs that could have been useful for some mutant they do not crash in the future, but re-trying all inputs for each mutant is expensive.

When only one CPU is available for fuzzing, the sequential vs. parallel nature of the approaches does not matter, but if many CPUs are available, then fuzzing many mutants at once is an obviously attractive proposition. While the total computing resources required to fuzz the same number of mutants are constant, that one approach is (embarrassingly) parallel is a significant advantage in modern multicore contexts. In fact, fuzzing mutants to some extent offers a simple solution to the problems of work division and communication overhead that trouble parallel fuzzing in general [**?**].

There are other minor variations. For instance, if the program under test has changed since the generation of any existing corpus, it may be useful to run a fuzzing stage on the target program to help seed the mutant fuzzing efforts, for the non-cumulative case. In the cumulative case, this is unlikely to be helpful, as early mutants will likely include near-equivalent programs, yielding the same effect with the added advantage of the opportunity offered by mutants.

III. Related Work

Given that getting past verification checks is one of the most common problems in fuzzing, (manually disabling verification checks is one of the most common proposals in practical [**?**] suggestions on improve the effectiveness of fuzzing) numerous previous researchers have tried to bypass such checks by patching the program itself. An early attempt to do this was Flayer [**?**] which provides a mechanism for

[2]These pruned inputs should be preserved and run against the actual target program, as they may represent uniquely detected faults.

instrumenting the program, altering the control flow, and stepping over function calls. The research also introduces a complementary fuzzer that makes use of Flayer for more effective fuzzing.

A similar approach was taken in TaintScope [**?**], which claims to be the first *checksum-aware* fuzzer. It detects checksum based integrity verification using branch profiling, and once found, it can bypass such checks by altering the control flow.

CAFA [**?**] is another fuzzer that uses taint analysis to detect the parts of the program that are involved in checksum based verification of input integrity. Once detected, it statically patches the program to bypass checksum verification of the input.

The most closely related work is the T-Fuzz approach [**?**], which focused specifically on removing sanity checks in programs in order to fuzz more deeply. Our approach is motivated in part by the desire to remove sanity checks, but uses a more general and lightweight approach. T-Fuzz used dynamic analysis to identify sanity checks, while we simply trust that program mutants will include many (or most) sanity checks. Moreover, when a sanity check is hard to identify, but implemented by a function call, statement deletion mutants may in effect remove it where T-Fuzz will not. Our approach also introduces changes that are not within the domain of T-Fuzz or the other fuzzers discussed above, e.g., changing conditions to include one-off values. Finally, T-Fuzz worked around the fact that inputs for the modified program are not inputs for the real program under test using a symbolic execution step, while we simply hand the inputs generated for mutants to a fuzzer and trust a good fuzzer to make use of these "hints" to find inputs for the real program, if they are close enough to be useful.

Mutation analysis has been used previously to detect anomalies in programs statically [**?**]. As in our approach, the program variants are produced using mutation analysis, but the idea here is to look for variants that are semantically equivalent, but better in some specific sense than the original.

Arguably, UBSAN is a program transformer that explicitly doesn't preserve all the program semantics (only the explicitly defined language semantics are preserved), and can improve fuzzing effectiveness. It detects undefined behavior by inserting crashes when such behavior is invoked.

Finally, mutants may prove to be effective against anti-fuzzing [**?**] techniques such as speed-bumps (a mutant could either remove the bump or simply decrease delay/wait loop parameters).

IV. Proposed Evaluation

In a full experimental evaluation, we will undertake to answer the following core research questions:

- **RQ1**: Does replacing time spent fuzzing a target program with time spent fuzzing mutants of the target program improve the effectiveness of fuzzing?
- **RQ2:** Does using prioritization improve the effectiveness of fuzzing with mutants? If so, which prioritizations perform best?
- **RQ3:** How do non-cumulative (parallel) and cumulative (sequential) mutant fuzzing compare?
- **RQ4:** For non-cumulative mutant fuzzing, is improving the corpus by first fuzzing the target program when it has changed worthwhile?

RQ1 is the overall question of whether any variant of fuzzing using mutants increases standard fuzzing evaluation metrics (unique faults detected and code coverage). **RQ2-RQ4** consider some of the primary choices to be made in implementing fuzzing mutants.

The experiments will be based on widely-used benchmarks, and conform to the standards proposed by Klees et. al [**?**], e.g., using 10 or more runs of 24 hours each in experimental trials. We will make every effort to identify and protect against the usual threats to validity in fuzzing experiments, by using a range of benchmark subjects and avoiding pitfalls such as measuring only crash counts bucketed crashes, rather than making an effort to identify actual distinct faults [**?**] (or using only crashes, not crashes and code coverage results).

One simplifying factor in experiments on this question is that, since the approach concerns only the choice of fuzzing targets and seeds, a single widely-used fuzzer, such as the latest version of AFL, is justified. It seems clear that the advantages provided by fuzzing mutants should be orthogonal to the varying features of AFL, AFLPlusPlus, libFuzzer, and other commonly used fuzzers. Even fuzzers that solve constraints to try to cover new branches (e.g., Eclipser [**?**]) do not attempt to solve branches not on the coverage frontier, such as `hard2` in our running example.

However, in order to check our assumption, we plan to perform a limited set of experiments on at least one fuzzer that is very different than the primary fuzzer used, e.g., using libFuzzer as a check on an AFL-based evaluation.

In addition to the primary research questions above, we plan to examine other practically important aspects of mutant fuzzing. For instance, while we expect most gains to be derived from using corpus inputs to help fuzz the target itself, we also believe, based on our preliminary experiments discussed below, that some bugs may be found only by fuzzing a mutant. How often does this happen on real programs, and why does it happen? One possibility of interest is that the coarse heuristics many fuzzers use to avoid storing duplicate crashes [**?**], [**?**] may sometimes discard non-redundant bugs, and that program mutants interact with AFL's heuristics to prevent this in some cases. We also plan to identify particular mutants that contributed to hitting hard-to-reach program paths, in order to better understand if there are patterns in useful mutants that can be predicted.

V. Preliminary Experiments

Table **??** shows results of fuzzing the `fuzzgoat` (https://github.com/fuzzstati0n/fuzzgoat) benchmark program for fuzzers, with and without using mutants to aid the fuzzing. We applied our basic technique, using both random and prioritized (by Universal Mutator) mutant selection, and using non-cumulative and cumulative mutant fuzzing. For non-cumulative mutant fuzzing, we did not perform an initial stage of fuzzing

TABLE I. RESULTS FOR PRELIMINARY EXPERIMENTS

	Distinct Faults			Statement Coverage			Branch Coverage		
Method	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
AFL on program only	3	5	4.2	79.86%	84.37%	81.73%	78.36%	81.35%	80.40%
AFL on random mutants, non-cumulative	**6**	**7**	**6.4**	80.04%	**84.90%**	81.70%	79.85%	**82.58%**	80.70%
AFL on random mutants, cumulative/sequential	**6**	**7**	6.2	80.21%	**84.90%**	81.77%	80.10%	82.34%	80.90%
AFL on prioritized mutants, non-cumulative	**6**	**7**	6.2	**81.25%**	84.37%	**82.39%**	**80.60%**	81.84%	81.20%
AFL on prioritized mutants, cumulative/sequential	**6**	**7**	6.2	**81.25%**	**84.90%**	83.16%	80.10%	**82.58%**	**81.39%**

on the target program. The best value(s) for each evaluation measure are highlighted in bold.

Each technique evaluated was used in 5 fuzzing attempts of 10 hours each. The baseline for comparison is the latest Google release of AFL (2.57b) on the `fuzzgoat` program for 10 hours, with no time spent in any effort other than fuzzing `fuzzgoat`. The other approaches apply the basic methods for using mutants described above, for five hours, then fuzz using the resulting corpus for another five hours. These approaches all spend a small fraction of the fuzzing budget restarting AFL and processing already-generated inputs (e.g., to make sure they don't crash the original program, even if they did not crash a mutant), rather than fuzzing either `fuzzgoat` or a mutant. The budget for fuzzing each mutant is fixed at five minutes, so only about 60 of the nearly 3,800 mutants of `fuzzgoat.c` can be fuzzed. For the first two mutant runs, these mutants were chosen randomly each time; the second two runs used a fixed set of mutants, based on the default mutant prioritization scheme provided by the Universal Mutator, with the option to prioritze all statement deletions above other mutants set to false. Coverage was measured using `gcov` and faults were determined by using address sanitizer to determine locations of memory access violations, and examining the traces to determine the distinct faults.

Fault detection was *uniformly better* for all mutant-based approaches than for fuzzing without mutants; the minimum number of detected faults was better than the maximum number of faults found without using mutants. Fault detection partly benefitted from crashes detected only during fuzzing of mutants. However, even ignoring these crashes, three of the mutant-baed efforts detected six distinct faults, while fuzzing without mutants never detected six faults. Means for the techniques without using crashes discovered during mutant fuzzing were, respectively (in the same order as the table): 4.8, 4.6, 5.0, and 5.0, still all higher than for fuzzing without mutants. Using the crashes from mutant fuzzing, every mutant-based effort detected all vulnerabilities in fuzzgoat of which we are aware.

Code coverage results were more ambigious, but the limited data suggests the prioritized mutant approaches may be more consistent in hitting hard-to-teach code than the other methods. In particular, the highest branch coverage numbers were all reached by prioritized mutant fuzzing, and the worst statement and branch coverage values were from fuzzing without mutants.

Coverage differences were not statistically significant by Mann Whitney U test, but bug count differences between all mutant-based methods and AFL without mutants were significant with p-value < 0.006. Differences in unique faults detected were not significant, when faults detected only during mutant fuzzing were discarded (though this is likely only due to the small sample size and range of values; p-values were around 0.2).

While it is clear that for this benchmark program, fuzzing mutants provides an advantage, it is also clear that distinguishing between variations of the basic approach is not possible without considerably more experimental data across more subjects.

Finally, we note that our experiments support our claim that the proposed technique is almost trivial to apply. We were able to implement mutant fuzzing in less than 30 lines of Python, and replacing AFL with another fuzzer would be trivial.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we propose that by fuzzing variations of a target program generated by a mutation testing tool, it may be possible to work around some fundamental limitations of coverage-driven fuzzing. For the most part, even when not effective, the technique proposed should be low-cost and at worst equivalent to fuzzing the target program itself for a somewhat smaller time. Our preliminary experiments show that fuzzing mutants is trivial to implement (and applies to any fuzzer of which we are aware) and effective for improving fault detection, at least for a non-trivial benchmark target program.

Future work, in addition to the performance of full experiments to evaluate the technique, would include exploring the effectiveness of using mutation selection methods and prioritization techniques in addition to those proposed here, and applying directed greybox fuzzing [**?**] to specifically target mutated code. Another possibility is to use Higher Order Mutants [**?**] to fuzz multiple mutants at once; however, this increases the chance that a critical mutant will be combined with a mutant that essentially destroys the program semantics, making it impossible to exploit.

ACKNOWLEDGEMENTS

A portion of this work was supported by the National Science Foundation under CCF-2129446. The authors would also like to thank our anonymous reviewers.