Registered Report: First, Fuzz the Mutants

ALEX GROCE, Northern Arizona University, United States
GOUTAMKUMAR TULAJAPPA KALBURGI, Northern Arizona University, United States
CLAIRE LE GOUES, Carnegie Mellon University, United States
KUSH JAIN, Carnegie Mellon University, United States
RAHUL GOPINATH, University of Sydney, Australia

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 effectiveness 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 results using Google's FuzzBench platform show that fuzzing mutants is trivial to implement and fuzzer-agnostic, but provides clear, statistically significant benefits in terms of branch coverage for a number of real-world benchmarks, using AFLplusplus and Honggfuzz as baseline fuzzers. One of the variants of our method, using heuristically chosen mutants, ranks first by both of the two standard measures of fuzzer effectiveness provided by FuzzBench.

CCS Concepts: • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

Additional Key Words and Phrases: fuzzing, mutation testing

ACM Reference Format:
Alex Groce, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, Kush Jain, and Rahul Gopinath. 2018. Registered Report: First, Fuzz the Mutants. In . ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Fuzzing is an essential tool for ensuring that software is robust, secure, and as error-free as possible [24]. 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:

Manuscript submitted to ACM

```
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 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.

[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.

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 [26]. 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 [17].

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 modifications 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.

This paper makes the following contributions:

- We propose a basic method for fuzzing using program mutants. The approach is easy to implement, and essentially applicable to any corpus-based fuzzer.
- We demonstrate, by a FuzzBench-based evaluation, that our approach is able to produce dramatic, statistically significant improvements over using state-of-the-art fuzzers on only the original program. In fact, one variation of our approach (using heuristically selected mutants) scores best by *both* of the primary FuzzBench evaluation methods, for a large set of benchmarks, compared to the AFLplusplus and Honggfuzz fuzzers. AFLplusplus and

Honggfuzz are state-of-the-art fuzzers that routinely place at or near the top of FuzzBench evaluations, showing that our method can improve even the best fuzzers now available.

- We provide a substantial independent commentary on the process of using FuzzBench, and the benefits and challenges of using FuzzBench in fuzzing research.

2 FUZZING THE MUTANTS, IN DETAIL

2.1 Mutation Testing

Mutation testing [7, 22, 26] 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 [2, 5, 28].

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 conditional, 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.

2.2 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.

2.2.1 Choosing the Mutants. First, there needs to be some source of mutants. For generating mutants, we use the regular-expression-based Universal Mutator [17] (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 [30] 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 10% 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 [16] proposed a scheme for ordering mutants for humans to examine, implemented in the Universal Mutator.

The mutant prioritization uses Gonzalez' Furthest-Point-First [13] (FPF) algorithm to *rank* mutants, as earlier work had used it to rank test cases for identifying faults [8]. 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 [17] tool's FPF metric d is the sum of a set of measurements. First, it adds a similarity ratio based on Levenshtein distance [21] 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:

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

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 [15]. However, the statistical properties that make purely random selection attractive in predicting mutation score are not as important for using mutants to aid fuzzing.

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.

Full Mutant Analysis, Continuous Mutant Analysis. Finally, for 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 be an alternative to fuzzing the target program with a rolling sequence of mutants (to at least those that ever generated useful inputs), in practice eliminating the clear demarcation 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.

2.2.2 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.

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

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, an approach that is (embarrassingly) parallel has 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 [32].

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.

3 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 [14] 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 [11] which provides a mechanism for 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 [31], 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 [23] 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 [27], 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 [3]. 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 [19] techniques such as speed-bumps (a mutant could either remove the bump or simply decrease delay/wait loop parameters).

4 EVALUATION METHOD

In our experimental evaluation, we undertook to answer the following core research questions:

- **RQ1**: Can replacing time spent fuzzing a target program with time spent fuzzing mutants of the target program, under at least one configuration, improve the effectiveness of fuzzing on average over a variety of target programs?
- **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:** How does impact of using mutants vary with the fraction of the fuzzing budget devoted to fuzzing mutants, given a good configuration based on results from **RQ2** and **RQ3**?
- **RQ5:** Are certain kinds of mutants generally more useful for fuzzing, or generally not useful? Are these patterns the same or different for new path discovery and novel bug detection?

RQ1 is the overall question of whether *any* variant of fuzzing using mutants explored increases standard fuzzing evaluation metrics (unique faults detected and code coverage, primarily). **RQ2**-**RQ5** consider some of the primary choices to be made in implementing fuzzing mutants, and help investigate why mutants are helpful, supposing an affirmative answer to **RQ1**.

RQ2 and **RQ3** and are straightforward, aiming to compare the effectivness of different basic approaches to using mutants in fuzzing (random or systematic selection of mutants, cumulative and non-cumulative approaches, respectively). Because exploring all combinations of mutant-fuzzing approach and budget would require a prohibitive number of experimental runs, we planned to focus the investigation of **RQ4** on answering the question, given a best-practice for using mutants in a baseline one-half-budget setting. The core idea of **RQ4** is to determine if, for most programs, one half of the budget is fuzzing mutants too long (such that diminishing returns have set in before this point, usually) or if further fuzzing of mutants, up to essentially the full budget, would continue to be useful, allowing for bypassing more difficult barriers to exploration.

Finally, for **RQ5**, while the universalmutator does not label mutations with operators, it is possible to label generated mutant source files with the regular expression rule that produced the mutant, and thus categorize mutants broadly, though not using exactly the standard operator definitions. Given such a categorization, we can determine if inputs produced by certain kinds of mutants tend to produce more new input paths that are valid for the original program, or produce more bugs only detected during mutant fuzzing. The second question is particularly interesting: while we expect most gains to be derived from using corpus inputs to help fuzz the target itself, we observe 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 [18, 29] may sometimes discard non-redundant bugs, and that program mutants interact with AFL's heuristics to prevent this in some cases. **RQ5** is more exploratory than the other research questions, in part aiming to discover which of the kinds of patterns proposed in the introduction appear most in practice. However, results for **RQ5** might also have more pragmatic importance, in informing refinement of mutant prioritization.

The set of experimental configurations shown below in the preliminary experiments gives an idea of the basic parameters to be varied. The set of fuzzers included in the real experiment will be larger and there will also be an exploration of different choices for the portion of the time budget devoted to fuzzing mutants. The experiments will be based on widely-used benchmarks, and conform to the standards proposed by Klees et. al [20], 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

Table 1. Results for preliminary experiments

	Distinct Faults			Statement Coverage			Branch Coverage		
Method	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Baseline (no mutants)									
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%
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%
Cumulative/sequential	**6**	**7**	6.2	**81.25%**	**84.90%**	83.16%	80.10%	**82.58%**	**81.39%**

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 [12] (or using only crashes, not crashes and code coverage results).

We used Google's FuzzBench [25] (https://google.github.io/fuzzbench/) to perform experiments. The lead of the FuzzBench team extended an invitation to use FuzzBench and assisted us in our efforts. Using FuzzBench allowed us to perform extensive experiments in a well-curated configuration, over a number of important fuzzers and benchmarks. Use of FuzzBench limited the potential for experimental error or poor experimental design choices, and should make future comparison with other fuzzing techniques easier.

We would also like to have evaluated our approach against T-Fuzz [27], but the last commit to the repository (https://github.com/HexHive/T-Fuzz) was in December of 2018, and our early efforts to get the tool to build and operate in an environment comparable to what will be used for other fuzzers was not promising (the versions of angr and other required tools are ancient and do not work with recent versions of Linux). Section 8 discusses our failure to produce useful T-Fuzz results.

5 PRELIMINARY EXPERIMENTS

Table 1 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 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 prioritize 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 benefited from crashes detected only during fuzzing of mutants. However, even ignoring these crashes, three of the mutant-based 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 ambiguous, but the limited data suggests the prioritized mutant approaches may be more consistent in hitting hard-to-reach 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.

6 FUZZBENCH EXPERIMENTAL RESULTS

6.1 RQ1: Basic Effectiveness

Figures 1 and 2 show that in some cases the impact of analyzing mutants can be a large improvement in code coverage. In all of our FuzzBench results, `um_random` indicates fuzzing including mutants, without mutant prioritization; similarly, `fuzzername_um_prioritize` would indicate fuzzing with baseline fuzzer `fuzzername` and prioritized mutants. We indicate deviations from our default 0.50 mutant-fuzzing budget by appending the fraction of the overall fuzzing budget spent on mutants, e.g., `honggfuzz_um_prioritize_75` would indicate fuzzing using Honggfuzz on prioritized mutants, where only the last 25% of the fuzzing budget was spent fuzzing the original program.

Bloaty (https://github.com/google/bloaty) is not a toy program; like all FuzzBench benchmarks, it is a real-world program, in this case with nearly 4,000 GitHub stars, and over 7,700 lines of code. Baseline AFLplusplus weak performance shows that it is not an easy target to fuzz effectively. Such dramatic improvements, with an effect size of hundreds of branches, with a Mann-Whitney U-test p-value of less than 0.001, and a 1.0 probability that mutant-based fuzzing outperforms the base fuzzer, were rare. However, the overall results support the utility of our approach, using AFLplusplus

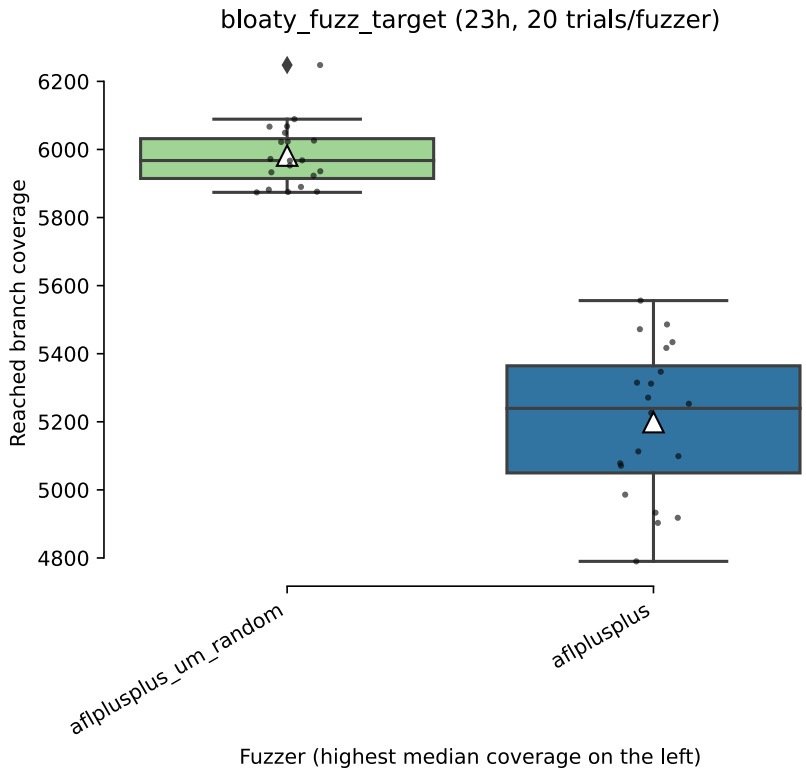

Fig. 1. AFLplusplus with and without (random) mutant fuzzing: final branch coverage

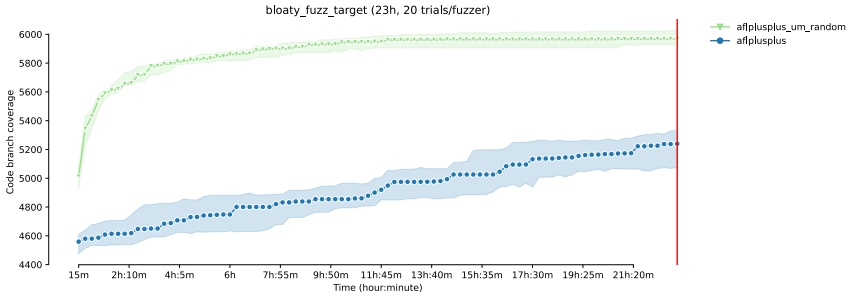

Fig. 2. AFLplusplus with and without (random) mutant fuzzing: branch coverage growth

(https://github.com/AFLplusplus/AFLplusplus) and Honggfuzz (https://github.com/google/honggfuzz) as our baseline fuzzers:

- Using the first of FuzzBench's standard evaluation measures, "based on the average of per-benchmarks scores, where the score represents the percentage of the highest reached median code-coverage on a given benchmark (higher value is better)", the best performing fuzzer in our experiments was AFLplusplus using prioritized

Fuzzer	Average Normalized Branch Coverage
aflplusplus_um_prioritize	97.91
aflplusplus	97.64
aflplusplus_um_random_75	97.60
honggfuzz_um_prioritize	97.45
honggfuzz_um_random	97.00
honggfuzz	96.97
aflplusplus_um_random	96.16
honggfuzz_um_prioritize_75	96.15
honggfuzz_um_random_75	95.29

Table 2. Ranking by Average Normalized Branch Coverage

Fuzzer	Average Fuzzer Rank
aflplusplus_um_prioritize	3.11
aflplusplus_um_random_75	3.27
aflplusplus_um_random	3.32
aflplusplus	3.52
honggfuzz	3.82
honggfuzz_um_prioritize	4.13
honggfuzz_um_prioritize_75	4.90
honggfuzz_um_random_75	5.00
honggfuzz_um_random	5.18

Table 3. Ranking by Average Fuzzer Rank

mutants, with a normalized score of 97.91. The next best fuzzer, AFLplusplus without mutants, had a score of 97.64. Honggfuzz with prioritized (97.45) or randomly chosen (97.0) mutants also performed better than baseline Honggfuzz (96.97) by this measure.

- Using the other standard FuzzBench evaluation measure, "average rank of fuzzers, after we rank them on each benchmark according to their median reached code-covereges (lower value is better)", AFLplusplus with prioritized mutants again scored best (average rank of 3.11), and all AFLplusplus mutant variations performed better than baseline AFLplusplus. Honggfuzz however, scored better than all mutant variants of Honggfuzz.

Tables 2 and 3 show full results for these core FuzzBench multi-benchmark evaluations, for the nine configurations we were able to run. The absence of non-cumulative variants, as well as of bug-based evaluations, is explained below (see Section 6.4).

Finally, Table 4 shows complete results for all benchmarks and fuzzers. **N/A** indicates that a particular run failed, usually due to build issues; note that this sometimes happened, due to the stress on the build assumptions discussed below, even for baseline fuzzers. The median and mean fuzzer statistics here are computed without the normalization used in the standard rankings shown above, but are still indicative of overall fuzzer performance, assuming no bias in missing results, and are of course valid for use in comparing pairs of fuzzers on a single benchmark. Again, methods using mutants scored highest for both median and mean branch coverage. To see full results, including statistical significance (most differences were significant at the $p < 0.05$ level), please see the complete FuzzBench generated report merging all of our experiments (https://github.com/agroce/fuzzing22report/blob/master/fuzzbench_report_10_17/report). The original experiment runs are available at the FuzzBench site (https://fuzzbench.com/reports/experimental/2022-10-13-um-final-

Benchmark	AFLplusplus				Honggfuzz				
		prioritize	random	random_75		prioritize	random	prioritize_75	random_75
bloaty_fuzz_target	79.20	94.73	90.39	94.79	98.40	97.03	95.58	93.02	92.94
curl_curl_fuzzer_http	N/A	N/A	N/A	N/A	99.01	N/A	N/A	N/A	N/A
freetype2-2017	90.33	91.58	N/A	91.33	95.19	95.65	N/A	96.06	96.02
harfbuzz-1.3.2	95.01	94.43	N/A	95.28	94.07	94.63	94.51	94.26	N/A
jsoncpp_jsoncpp_fuzzer	99.04	N/A	N/A	N/A	99.42	99.81	N/A	99.81	N/A
lcms-2017-03-21	91.94	91.91	92.03	N/A	66.81	88.32	N/A	71.31	N/A
libjpeg-turbo-07-2017	97.71	97.99	98.04	97.75	N/A	96.08	92.67	96.13	96.18
libpcap_fuzz_both	89.47	N/A	71.78	N/A	86.96	87.31	N/A	85.28	85.43
libpng-1.2.56	93.02	N/A	N/A	93.10	95.05	97.17	95.36	95.05	97.97
libxml2-v2.9.2	86.09	N/A	N/A	N/A	91.23	N/A	90.98	90.72	90.51
libxslt_xpath	98.84	N/A	N/A	N/A	97.78	N/A	N/A	N/A	N/A
mbedtls_fuzz_dtlsclient	78.28	78.62	N/A	78.71	77.18	76.69	77.00	77.01	N/A
openssl_x509	99.97	99.89	99.89	99.85	99.54	99.48	N/A	99.43	N/A
openthread-2019-12-23	92.93	N/A	N/A	N/A	92.12	79.22	91.09	78.73	77.38
php_php-fuzz-parser	98.76	99.10	99.07	99.18	99.04	98.89	98.78	98.56	98.58
proj4-2017-08-14	86.82	89.33	N/A	89.50	96.11	96.43	N/A	95.34	N/A
re2-2014-12-09	99.32	98.60	98.56	98.58	98.52	98.58	N/A	98.54	N/A
sqlite3_ossfuzz	94.03	83.39	81.42	84.56	81.49	81.12	80.73	81.02	79.07
systemd_fuzz-link-parser	100.00	100.00	100.00	100.00	N/A	99.15	99.15	99.15	N/A
vorbis-2017-12-11	98.67	98.67	98.82	98.67	97.65	97.53	97.49	97.61	N/A
woff2-2016-05-06	97.62	97.78	97.78	97.66	96.18	96.39	N/A	96.51	N/A
zlib_zlib_uncompress_fuzzer	97.98	N/A	N/A	94.90	N/A	97.98	N/A	98.51	97.45
Median	95.01	96.26	**98.04**	95.28	96.11	96.43	94.51	95.70	94.48
Mean	93.57	94.00	93.43	**94.26**	92.72	93.55	92.12	92.10	91.15

Table 4. Full Median Relative Branch Coverage Results for FuzzBench Benchmarks

1/index.html, https://fuzzbench.com/reports/experimental/2022-10-13-um-final-2/index.html, https://fuzzbench.com/reports/experimental/2022-10-13-um-final-4/index.html, and https://fuzzbench.com/reports/experimental/2022-10-13-um-final-5/index.html).

While our results are limited, we note that mutants seemed to have better impact on AFLplusplus than on Honggfuzz. Even so, out of the 16 benchmarks where we have results for both base Honggfuzz and and Honggfuzz over prioritized mutants, the prioritized approach performs better in terms of median branch coverage for 9 benchmarks. Moreover, the impact was sometimes large: for the lcms-2017-03-21 benchmark, for example, baseline Hongfuzz only covered a median of 66.81% of the best achieved branch coverage; using prioritized mutants improved this to a median of 88.32% of best branch coverage, with a p-value of < 0.05.

6.1.1 Individual Benchmark Fuzzing Impacts. Restricting our attention to comparisons of AFLplusplus with AFLplusplus over randomly chosen mutants for half the fuzzing budget, we can observe that the effect of fuzzing mutants can be dramatic, and can vary considerably. We focus on randomly chosen mutants despite the overall superior performance of prioritized mutants in order to show that even with random selection of mutants, the variance for fuzzing can in many cases be no greater than for fuzzing without mutants. Full results for this comparison can be examined in the FuzzBench report repository: https://fuzzbench.com/reports/experimental/2022-10-13-um-final-1/index.html.

Figures 3 and 4 show that the impact of mutant fuzzing is not always, as with the bloaty benchmark, evident during the mutant-fuzzing phase. Here, the (consistent) advantage over AFLplusplus only appears after fuzzing has swiched to the original target program. Presumably, seeds generated by some mutants "pay off" during the final run of original-target fuzzing. Again we note that many mutants are likely essentially equivalent in their impact on fuzzing, since the confidence intervals on branch coverage are tight despite the use of randomly selected, likely non-overlapping, mutants in these runs and the bloaty runs.

Finally, it is not always the case that mutants having a strong impact on fuzzing pays off in the long run. Figures 5 and 6 show that fuzzing mutants can provide a statistically significant advantage over a baseline fuzzer early in the run, but eventually ends up performing significantly worse over a longer period. We are not sure what causes such results:

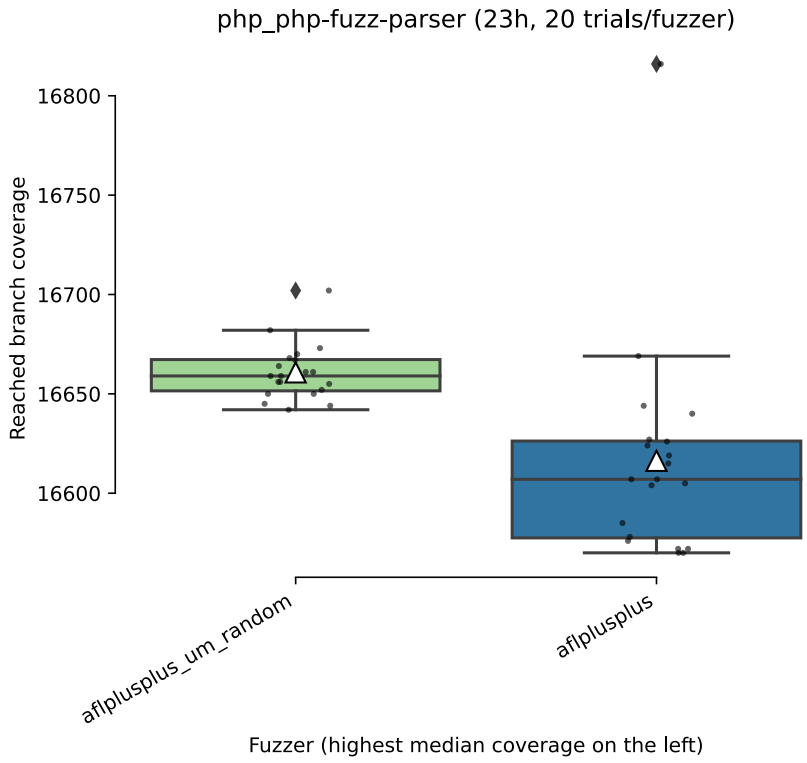

Fig. 3. AFLplusplus with and without (random) mutant fuzzing: final branch coverage

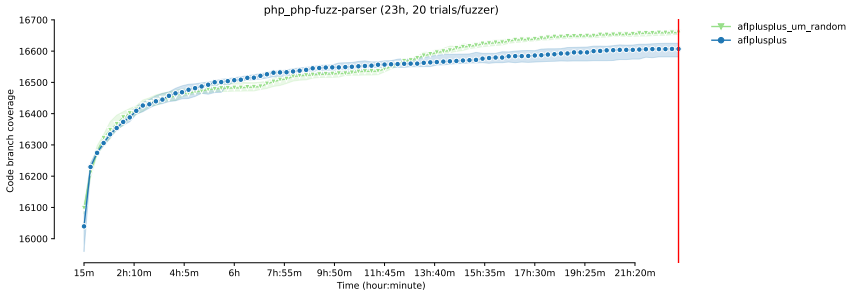

Fig. 4. AFLplusplus with and without (random) mutant fuzzing: branch coverage growth

one possibility is that many mutants are harmful, and the useful mutants that cause the early advantage help with overcoming fuzzing barriers that also can be defeated simply by using a good fuzzer for a longer period.

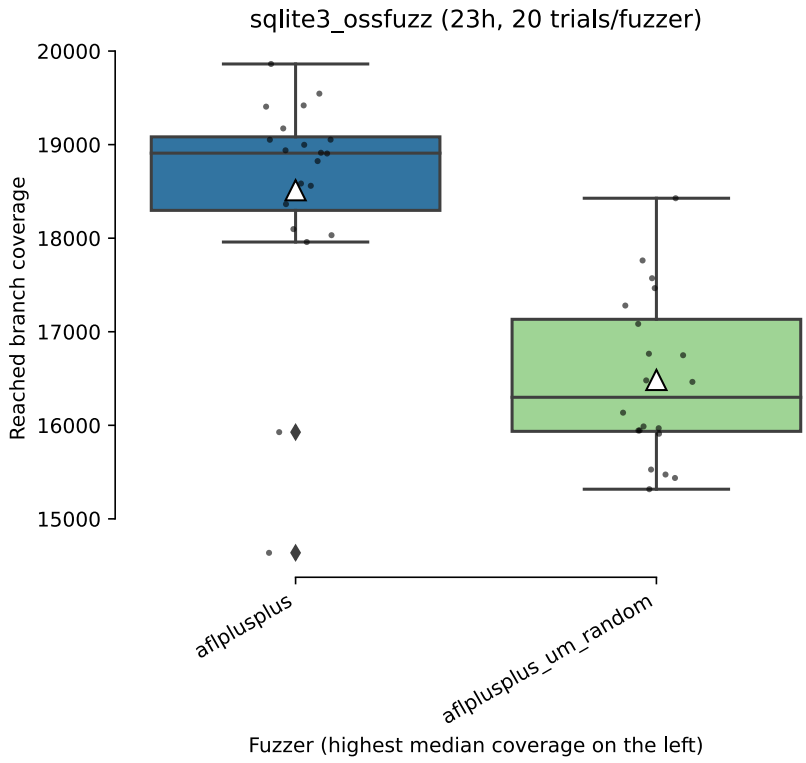

Fig. 5. AFLplusplus with and without (random) mutant fuzzing: final branch coverage

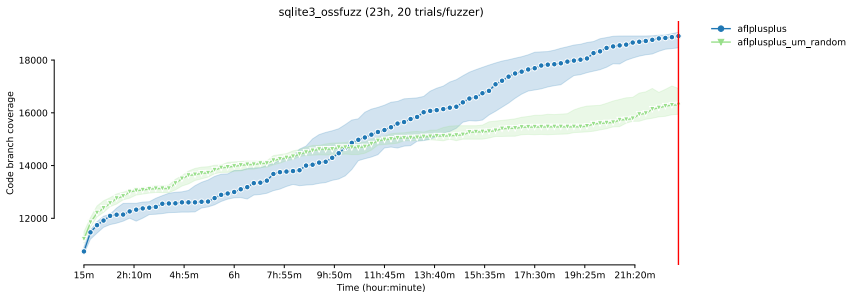

Fig. 6. AFLplusplus with and without (random) mutant fuzzing: branch coverage growth

6.2 RQ2: Effects of Mutant Prioritization

In all cases, the prioritized variants of the approach scored better than using randomly chosen mutants. We would need more fuzzers to be sure that this effect persists across all fuzzers, but our results suggest that even a simple prioritization designed for human use is also effective for fuzzing purposes.

6.3 RQ4: Effects of Varying Mutant-Fuzzing Budget

We would need more fuzzers and more budget allocations to make any claims (and at minimum data for an AFLplusplus prioritized longer mutant phase). By both FuzzBench measures, AFLplusplus random mutants were improved by running longer, but for Honggfuzz, using 75% of the time for mutants performed worse by average normalized branch coverage.

6.4 The FuzzBench Experience

Overall, our experience confirms that FuzzBench is an important contribution to fuzzing research. The experiments are conducted based on established best practices for fuzzing evaluation [20]. Because of the large number of complex benchmarks included, differences in fuzzers can be demonstrated using only coverage, a metric that is not perfect [12] but that is also likely to be much more reliable than most bug-count based results on real-world subjects [8, 12]. In addition to the kinds of best practices proposed in Klees et al., FuzzBench reports include sophisticated statistical analysis, in order to help researchers report their results honestly. These go beyond normative software engineering statistical methods [4] and include methods from evaluation of machine learning algorithms [10] that are good fits for fuzzing research. We found the FuzzBench team, especially Jonathan Metz, extremely helpful, and for the most part the development of our experiment code was made easy by the FuzzBench infrastructure.

However, as the incomplete results above show, FuzzBench is not yet an ideal platform for fuzzing research. In particular, the FuzzBench model made a few assumptions that resulted in great difficulties for us:

(1) FuzzBench assumes it is reasonable to split the experiment into a *build* stage that produces binaries for benchmarks and a *fuzzing* stage that simply consumes binaries.
(2) FuzzBench further assumes that *one* build stage per benchmark/fuzzer pair is all that is required.
(3) FuzzBench's approach to making the build code for a fuzzer (which may need to, e.g., swap in a new C/C++ compiler) benchmark-agnostic imposes a high cost for builds: each build of a binary builds the entire benchmark, including potentially a large amount of dependency code that needs to be instrumented, from scratch. For many projects this takes time measured not in seconds but in minutes.

For most fuzzers, where the requirements for the fuzzing stage are one or at most two binaries per benchmark, this works well. However, because the process prevents us from building binaries for mutants during fuzzing (there is no support for this, and if it were made possible, the need to build completely from scratch for even tiny changes to one file would be unrealistically expensive), and we require hundreds of binaries, our experiments stressed the FuzzBench infrastructure in ways it had not previously been stressed. Jonathan Metz commented, near the deadline for this submission, "anyway, y0ur paper did a great job exposing some soft places in fuzzbench's design. I just hope we can handle these before the deadline." Among other issues exposed, disk space and limits on debugging logs (to find build issues), plus the problem of pre-emptive cloud instances causing many restarts for experiments with long build times, all introduced long delays in obtaining experimental results or determining why experiments did not run as expected. Even after the FuzzBench team and ourselves spent over a month (8/25/22 - 10/15/22) debugging our build scripts to work around issues, and in a few cases Jonathan modified FuzzBench parameters or code, some runs and builds failed, and the turnaround for experiments was many times the normal FuzzBench wait.

Due to these issues, our results do not include:

- **A bug-based evaluation**: This requires significant time investment including manual labor by the FuzzBench team, after an experiment is complete, and our experiments only finished at the moment of the deadline. We

expect to have such results by publication date, but for a large-scale experiment such as this, the exact timeline is hard to predict.

- **All desired configurations for mutant-based fuzzing** in particular, the non-cumulative approach, which requires more disk space, is not included. Exhaustion of disk space during build or fuzz stages was a continual problem for our experiments, so we were advised to not include this approach until those issues can be better resolved. We also did not extensively experiment with different choices for how much of the fuzzing budget to devote to mutants (**RQ4**); the only variation included is one where 75%, rather than the default 50%, of the budget is devoted to mutants. Similarly, our investigation of **RQ2** is limited to exploring the one prioritization defined by the universal mutator.
- **All desired fuzzer baselines**: we only compared against AFLplusplus and Honggfuzz. These are consistently among the best fuzzers in FuzzBench results, and are both widely used in industry and academia. However, we would very much like to also examine, at minimum, Eclipser, libFuzzer, and other well-known fuzzers, to see how much the usefulness of our approach varies by benchmark vs. by fuzzer.
- **FuzzBench based data for RQ5**: it may be possible to record information to examine **RQ5** in FuzzBench, but we were unable to attempt this due to the problems discussed above; moreover, the time spent getting *any* useful results for the core research question of effectiveness prohibited us from debugging experiments that attempted to answer this question using LAVA-M benchmarks. We therefore only report RQ5 results for our preliminary experiments.

Beyond these omissions, one of our final set of five experiments (combined into a single report with a single coverage normalization) did not actually run, due to as yet undiagnosed build issues We expect to have results for at least some of these omissions in the near future, but cannot guarantee this for the first two items, since these require efforts by the FuzzBench team beyond simply running additional experiments for us.

It is important to note that *none of these problems are an inherent part of our approach.* Normally, mutant binaries would be generated on the fly during fuzzing; using incremental builds after modifying a single source line in one file, such changes would usually require a very small part of the fuzzing budget. This is how our preliminary results were produced, and how the scripts we plan to release for a standalone tool implementing our method work. Unfortunately, FuzzBench by its nature precludes the fuzzing stage from seeing benchmark source, and forces us to use a cumbersome process that breaks various FuzzBench assumptions.

We further note that the way FuzzBench assembles source files for benchmarks meant that we often ended up mutating code that was not an important part of the benchmark target, but was part of a dependency that was compiled with instrumentation. In practice, in real world applications, a user would provide a list of relevant source files, which might also limit mutation to critical functionality (e.g., avoiding mutating logging code or user interface code not relevant to a fuzzing harness).

Finally, despite these problems, we encourage other researchers to use FuzzBench in their evaluations! The primary source of our difficulties, that most fuzzers only use one target binary while we expect to produce and use hundreds or even thousands of binaries, means that *most researchers will not face the challenges we faced.* On the other hand, FuzzBench introduces a degree of standardization to fuzzing comparisons and, as discussed below, mitigates important threats to validity. The readability and thoroughness of FuzzBench reports, moreover, made it possible to write up our conclusions, including quality graphics and statistical analyses, in less than a day once final results were available. Finally, we note that efforts to improve FuzzBench (e.g., the impacts our experiments have had and will have to build

infrastructure choices) are available to all researchers, rather than being lost as each team essentially re-invents the wheel.

7 RQ5: USEFUL MUTANT TYPES

Because we were unable to inspect FuzzBench individual mutant impacts, or get LAVA-M based experiments (chosen for direct comparison with T-Fuzz reported results) running in time, due to the difficulty of getting core FuzzBench results, we do not have extensive results to report for this question. However, we were able to examine the results in preliminary experiments in detail, and see which individual mutants and mutant types contributed most to detecting fuzzgoat bugs not found by at least one of the 10 hour AFL runs. That is, we looked at cases where the five-minute fuzzing of a mutant found a crash signature that did not appear in at least one of the full AFL runs. We scored mutants according to how often they contributed to such "novel bug discovery" and our findings were suggestive:

- Overall, only two types of mutant were unusually common among "bug-finding" mutants: changing `if` conditionals to false (i.e., removing an entire block of code under an `if`), and adding a `break` to a loop (i.e., removing an entire block of code at the end of a loop). This suggests that *large-scale* code deletions may provide extra utility. Surprisingly, statement deletions were not present in the "most useful" mutant sets at all. This may relate to observations made during use of code deletion for resource adaptation, where computed reductions tended to involve large chunks of code [9]; perhaps (somewhat unintuitively) larger deletions are likely to remove fuzzer blocks without compromising functionality to such a degree that behavior no longer resembles the original program.
- On the other hand, these types of mutants accounted for a minority of useful mutants, and the remaining mutants had no obvious pattern, and even included such unlikely changes as replacing a JSON error string with the empty string (which results in a smaller output buffer size, and perhaps faster fuzzing throughput, is our only speculation as to its utility).

Obviously, results over larger and more realistic benchmarks and for full experiments are needed to make strong claims, but it is at least not clear that any particular types of mutants could be safely omitted from the mutation process, given the unusual ways in which mutants contributed to finding rare fuzzgoat bugs.

8 COMPARISON WITH T-FUZZ

Unfortunately, we were unable to compare our approach with T-Fuzz [27]. We attempted to contact the authors, and did contact the author of a more recent fork of the T-Fuzz code, Yoshiki Takashima, who directed us to contact Maverick Woo, who might be able to shed more light on running T-Fuzz. Both confirmed that they had been unable to get T-Fuzz to run properly after attempting to do so.

We also explored using a Docker image for T-Fuzz (https://hub.docker.com/r/zjuchenyuan/tfuzz) that contained numerous comments on the original experiments, including some notes on problems with those experiments, developed by the authors of T-Fuzz. Unfortunately, all T-Fuzz benchmarks, after about ten minutes of fuzzing, with no results produced, terminated with the following form of error message:

```
  Traceback (most recent call last):
  File "./TFuzz", line 64, in <module>
    main()
  File "./TFuzz", line 55, in main
    tfuzzsys.run()
  File "/T-Fuzz/tfuzz/tfuzz_sys.py", line 204, in run
    shutil.copyfile(self.fuzzing_program.program_path, transformed_program_path)
  File "/usr/lib/python2.7/shutil.py", line 69, in copyfile
    raise Error("`%s` and `%s` are the same file" % (src, dst))
shutil.Error: `/T-Fuzz/workdir_uniq/uniq_5/uniq` and `/T-Fuzz/workdir_uniq/uniq_5/uniq` are the same file
```

Substantial inspection of the source code and further investigation did not result in any way to work around this error, and notes on the Docker suggested that recreating the original experiments might be difficult. Our failure to duplicate T-Fuzz results (or run T-Fuzz at all) does demonstrate a key difference between our approach and that of T-Fuzz: while conceptually similar, at a certain level, our approach relies on reliable off-the-shelf components (existing fuzzers and mutation testing tools) and can be implemented in a small script with hooks for fuzzing and target mutation/build. T-Fuzz, on the other hand, relies on a large number of non-robust components and is clearly subject to "bit rot" in fairly short order.

9 THREATS TO VALIDITY

The primary threat to validity here is that our experiments are incomplete. For some research questions, this essentially makes our results at present not useful. For the primary research question, **RQ1**, however, there seems to be little doubt, assuming the fuzzer evaluation best-practices encapsulated in FuzzBench are reasonable, that our approach has an overall significant positive effect on branch coverage. The major threat to validity is that this large advantage in some cases in branch coverage may not translate to an advantage in bug detection. However, this seems unlikely. Previous FuzzBench experiments have generally shown a very good correlation between bug and coverage based evaluations. For example, consider the two reports https://fuzzbench.com/reports/2022-04-19-bug/index.html and https://fuzzbench.com/reports/2022-04-19/index.html. These are coverage and bug-based evaluations for the same set of fuzzers and benchmarks on the same date. There is some difference in exact rankings, but the overall winners (Honggfuzz and AFLplusplus) are unchanged, and most individual fuzzer pairings are also unchanged.

Some threats to validity present in many fuzzing experiments are likely less relevant to our results: implementation bugs in the evaluation framework in particular are much less likely, due to the developer support and effort expended by Google on FuzzBench, as well as the probability that any major bugs would be detected over the course of the large number of consequential fuzzer evaluations performed using FuzzBench.

10 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 experiments show that fuzzing mutants is trivial to implement (and applies to any fuzzer of which we are aware) and effective for improving branch coverage across a large set of benchmarks, evaluated using Google's FuzzBench platform. Our methods, by the two primary FuzzBench ranking approaches, scored highest, and improvements over existing high-quality fuzzers were sometimes large.

Future work, in addition to the completion of full FuzzBench experiments to evaluate the technique and answer all research questions, would include exploring the effectiveness of using mutation selection methods and prioritization

techniques in addition to those proposed here, and applying directed greybox fuzzing [6] to specifically target mutated code. Another possibility is to use Higher Order Mutants [33] 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.

Finally, our difficulties with FuzzBench build restrictions turned out to possibly be beneficial, in that they inclined us to consider the possibility of *not using source-level mutants at all*. The existence of binary-based mutation tools such as SN4KE [1] suggests that mutation could be applied to a single target binary, instead of to source code. This has a notable advantage beyond making FuzzBench experiments much easier, in that, like QEMU-based fuzzers such as the AFL QEMU mode and the original Eclipser, it works for fuzzing binaries without source code. The primary disadvantage of binary or bytecode level mutation, that humans cannot easily understand such mutants or their implications, is irrelevant: the fuzzer does not need to understand source-level impacts of semantic changes. The changes useful for fuzzing are obviously equally available at the binary/assembly level as at the source level, and in fact binary-level mutants may introduce useful operators not easily implemented at the source level. The largest obstacle to this approach is the lack of maturity in binary-level mutation tools compared to source-level mutation tools, for C and C++ code, and, most importantly, the challenge of avoiding mutating fuzzer instrumentation itself. The latter problem does not apply to QEMU-based fuzzing, however. Moreover, we speculate that fuzzer instrumentation is typically much less than half of an instrumented binary's code size, often as little as 25%, and so most mutants at binary level would modify relevant target program code. Using knowledge of fuzzer instrumentation implementation it might also be easy to avoid mutating instrumentation. Finally, it is possible that some mutants to instrumentation could be beneficial, and that most would be harmless, so in some sense the difference between target code and instrumentation code mutants is less important than might at first seem to be the case.

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, as well as Yoshiki Takashima and Maverick Woo; and we would especially like to thank the FuzzBench team, and Jonathan Metz in particular.

REFERENCES

[1] Mohsen Ahmadi, Pantea Kiaei, and Navid Emamdoost. 2021. Sn4ke: Practical mutation testing at binary level. *arXiv preprint arXiv:2102.05709* (2021).

[2] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. 2017. Applying Mutation Analysis on Kernel Test Suites: an Experience Report. In *International Workshop on Mutation Analysis*. 110–115.

[3] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Paolo Vavassori. 2017. A novel use of equivalent mutants for static anomaly detection in software artifacts. *Information and Software Technology* 81 (2017), 52–64.

[4] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[5] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[7] Timothy Budd, Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1979. *Mutation analysis*. Yale University, Department of Computer Science.

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Programming Language Design and Implementation*. 197–208.

[9] Arpit Christi, Alex Groce, and Rahul Gopinath. 2017. Resource Adaptation via Test-Based Software Minimization. In *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 61–70. https://doi.org/10.1109/SASO.2017.15

[10] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* 7 (dec 2006), 1–30.

[11] Will Drewry and Tavis Ormandy. 2007. Flayer: Exposing Application Internals. In *First USENIX Workshop on Offensive Technologies, WOOT '07, Boston, MA, USA, August 6, 2007*, Dan Boneh, Tal Garfinkel, and Dug Song (Eds.). USENIX Association. https://www.usenix.org/conference/woot-07/flayer-exposing-application-internals

[12] Miroslav Gavrilov, Kyle Dewey, Alex Groce, Davina Zamanzadeh, and Ben Hardekopf. 2020. A Practical, Principled Measure of Fuzzer Appeal: A Preliminary Study. In *20th IEEE International Conference on Software Quality, Reliability and Security, QRS 2020, Macau, China, December 11-14, 2020*. IEEE, 510–517. https://doi.org/10.1109/QRS51102.2020.00071

[13] Teofilo F. Gonzalez. 1985. Clustering to Minimize the Maximum Intercluster Distance. *Theoretical Computer Science* 38 (1985), 293–306.

[14] Google. 2022. Efficient Fuzzing Guide. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/testing/libfuzzer/efficient_fuzzing.md.

[15] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation Reduction Strategies Considered Harmful. *IEEE Trans. Reliab.* 66, 3 (2017), 854–874. https://doi.org/10.1109/TR.2017.2705662

[16] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qi Hong Chen. 2021. Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis. In *IEEE International Conference on Software Quality, Reliability, and Security*.

[17] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-expression-based Tool for Multi-language Mutant Generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/3183440.3183485

[18] Josie Holmes and Alex Groce. 2020. Using mutants to help developers distinguish and debug (compiler) faults. *Softw. Test. Verification Reliab.* 30, 2 (2020). https://doi.org/10.1002/stvr.1727

[19] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-fuzzing techniques. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1913–1930.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

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

[22] Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[23] Xiaolong Liu, Qiang Wei, Qingxian Wang, Zheng Zhao, and Zhongxu Yin. 2018. CAFA: A Checksum-Aware Fuzzing Assistant Tool for Coverage Improvement. *Security and Communication Networks* 2018 (2018).

[24] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. J. Schwartz, and M. Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 01 (oct 2021), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[25] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1393–1403.

[26] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056

[28] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521

[29] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 612–622. https://doi.org/10.1145/3238147.3238200

[30] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 363–378. https://doi.org/10.1145/3314221.3314589

[31] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[32] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. 2021. Facilitating Parallel Fuzzing with Mutually-Exclusive Task Distribution. In *International Conference on Security and Privacy in Communication Systems*. Springer, 185–206.

[33] Chu-Pan Wong, Jens Meinicke, Leo Chen, João P Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently finding higher-order mutants. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1165–1177.