

# First, Fuzz the Mutants

Michael Shell  
Georgia Institute of Technology  
someemail@somedomain.com

Homer Simpson  
Twentieth Century Fox  
homer@thesimpsons.com

James Kirk  
and Montgomery Scott  
Starfleet Academy  
someemail@somedomain.com

**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 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. This 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

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 satisfying `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`, such an input will be discarded, because execution never reaches the implementation of `hard2` unless `hard1` has already been “solved.” 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. 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!

If we fuzz a program without 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 essentially impossible, to generate for the original program using the fuzzer.

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. Finally, predicting which program variants will aid fuzzing seems inherently hard. In this case, removing a statement was extremely useful; in other cases breaking out of a loop before it fails a check 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 [11]. 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 number 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.

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](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.

## II. FUZZING THE MUTANTS, IN DETAIL

### A. Mutation Testing

Mutation testing [11], [3], [10] is an approach to evaluating and improving tests. MObutation 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 [1], [13], [2].

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.

For generating mutants, we use the Universal Mutator [7] (<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++, Python, and Solidity code.

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 approach is to split a given time-budget for fuzzing in two. First, fuzz the mutants. Then, collect an input corpus from that fuzzing, and fuzz the target program as usual, but for half the desired 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*: For most programs, reasonable (e.g., 24 hour) fuzzing budgets, and approaches to fuzzing mutants discussed below, it is impossible 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: 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, while we cannot predict the best mutants, it might be reasonable to try to diversify the mutants selected using some kind of prioritization. In particular, in our recent work on using mutants to evaluated static analysis tools [6], we proposed a scheme for ordering mutants for humans to examine.

The mutant prioritization uses Gonzalez’ Furthest-Point-First [5] (FPF) algorithm to *rank* mutants, as earlier work had used it to rank test cases for identifying faults [4]. 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 [7] tool’s FPF metric  $d$  is the sum of a set of measurements. First, it adds a similarity ratio based on Levenshtein distance [9] 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 in 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.

2) *Using the Mutants*: The second key choice is how to use the chosen mutants. Assuming a fixed budget per mutant, the most basic choice is whether to fuzz each mutant “from scratch” (presumably 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) Mutants have some of the benefits of fuzzing with mutants, so hitting a key location that has been mutated may be more likely.
- 2) The final corpus from the last-fuzzed mutant will contain few redundancies, reducing processing or fuzzer startup time for the target.

On the other hand, it 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. These should be kept, as they may represent uniquely detected faults. 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, but re-trying all inputs for each mutant is expensive.

### III. 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?

- **RQ3**: How do non-cumulative (parallel) and cumulative (sequential) mutant fuzzing compare?

**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** and **RQ3** 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 [8], e.g., using 10 or more runs of 24 hours each in experimental trials. 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.

### IV. PRELIMINARY EXPERIMENTS

Table IV shows results of fuzzing the `fuzzgoat.c` (<https://github.com/fuzzstati0n/fuzzgoat>) benchmark program for fuzzers, with and without using mutants to aid the fuzzing. Each technique was used in 5 fuzzing attempts of 10 hours each. The baseline for comparison is the latest Google release of AFL 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. 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. Code coverage results were more ambiguous, but the limited data suggests the prioritized mutant approaches may be more consistent in hitting hard-to-teach code than the other methods.

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  $< 0.006$ .

### V. RELATED WORK

The most closely related work is the the T-Fuzz approach [12], 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

Method	Distinct Faults			Statement Coverage			Branch Coverage		
	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%

TABLE I. RESULTS FOR PRELIMINARY EXPERIMENTS

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, 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.

## VI. CONCLUSIONS

### REFERENCES

- [1] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, “Applying mutation analysis on kernel test suites: an experience report,” in *International Workshop on Mutation Analysis*, March 2017, pp. 110–115.
- [2] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, “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)*, 2021, pp. 268–277.
- [3] T. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, *Mutation analysis*. Yale University, Department of Computer Science, 1979.
- [4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, “Taming compiler fuzzers,” in *Programming Language Design and Implementation*, 2013, pp. 197–208.
- [5] T. F. Gonzalez, “Clustering to minimize the maximum intercluster distance,” *Theoretical Computer Science*, vol. 38, pp. 293–306, 1985.
- [6] A. Groce, I. Ahmed, J. Feist, G. Grieco, J. Gesi, M. Meidani, and Q. H. Chen, “Evaluating and improving static analysis tools via differential mutation analysis,” in *IEEE International Conference on Software Quality, Reliability, and Security*, 2021.
- [7] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, “An extensible, regular-expression-based tool for multi-language mutant generation,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 25–28. [Online]. Available: <http://doi.acm.org/10.1145/3183440.3183485>
- [8] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: <https://doi.org/10.1145/3243734.3243804>
- [9] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.
- [10] R. J. Lipton, R. A. DeMillo, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [11] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, “Mutation testing advances: an analysis and survey,” in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.
- [12] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-Fuzz: Fuzzing by program transformation,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.
- [13] G. Petrović and M. Ivanković, “State of mutation testing at google,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–171. [Online]. Available: <https://doi.org/10.1145/3183519.3183521>