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 effectivness of fuzzing, even if it means less time is spent fuzzing the actual target system. In particular, we claim that fault detection and code coverage can be improved by splitting fuzzing resources between the SUT and *mutants* of the SUT. Spending half of a fuzzing budget fuzzing mutants, and then using the seeds generated to fuzz the SUT can allow a fuzzer to explore more behaviors than spending the entire fuzzing budget on the SUT. The approach works because fuzzing most mutants is "almost" fuzzing the SUT, but may change behavior in ways that allow a fuzzer to reach deeper program behaviors. 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 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`, 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 statemeent 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.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23xxx
www.ndss-symposium.org

Program *mutants* provide such variants, by design [6]. 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) 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 [6], [3], [5] 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 [1], [8], [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 conditonal, or replacing arithmetic and relational operations (e.g., changing + to − or == to <=), are very widely used.

For generating mutants, we use the Universal Mutator [4] (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.

B. Fuzzing: Two Key Decisions

1) Choosing the Mutants:

2) Using the Mutants:

III. Proposed Evaluation

- **RQ1**:

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 if 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 ambigious, 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 [7], 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 analsyis 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, 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 exeuction step, while we simply hand

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

TABLE I. RESULTS FOR PRELIMINARY EXPERIMENTS

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] 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 Proceeedings*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 25–28. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183485

[5] 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.

[6] 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.

[7] 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.

[8] 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