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."

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

However, if we first fuzz the mutant that deletes 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*. 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 real crashing input for the original program. This is a very simple example of a case where fuzzing a similar program can produce inputs that 1) help fuzz the actual program under test and 2) where those inputs are much harder to generate for the original program.

II. Fuzzing the Mutants, in Detail

A. Mutation Testing

Mutation testing [5], [3], [4] 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], [7], [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.

III. Preliminary Experiments

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.

	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

IV. RELATED WORK

The most closely related work is the the T-Fuzz approach [6], 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 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.

V. 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] 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.

[5] 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.

[6] 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.

[7] 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