FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques

Andrea Fioraldi[1,2] and Luigi Paolo Pileggi[1]

[1]Sapienza University, Rome, Italy
{fioraldi.1692419, pileggi.1691249}@studenti.uniroma1.it

[2]EURECOM, Sophia Antipolis, France
fioraldi@eurecom.fr

1. Introduction

Feedback-driven Fuzz Testing is a family of techniques to automatically uncovers bugs in software.

Due to its effectiveness, much more efficient than other software testing techniques like *Symbolic Execution* [1] [2], the research in this field is flourishing and several different techniques were developed to improve fuzz testing, both from academy and industry.

The evaluation and the comparison of these techniques, however, is a debatable matter [3].

A common proxy is the comparison of the code coverage reached over time by each fuzzer, due to the fact that a fuzzer cannot find a bug if it does not explore at least the vulnerable code segment. Another widely used metric is found bugs over time, but a bug can be found just thanks to randomism and this makes the evaluations very prone to overfitting.

The data collected using these metrics are often representable using a simple time-based graph that shows the evolution of the fuzzing algorithm.

This approach is useful for an immediate basic comparison between two or more techniques, an analyst have to just see which technique reaches more coverage in less time, but does not reveal the properties of a fuzzer regards specific types of programs states.

For instance, a technique can be better than another in exploring some types of program states and in the same time reaching less code coverage. The technique will not cover the bugs in the unexplored code of course, but it may uncover bugs in the program points that it can better explore. An example of such technique is the directed fuzzer towards sanitizers violations by Österlund et al. [4].

We propose FUZZSPLORE, a tool that allows an analyst to manually explore the evolution of different fuzzing techniques on the same target program.

The main insight that an user can get using the tool are:

1) The ability of a fuzzer to generate clusters of inputs that are correlated in terms of covered program points;
2) The ability of a fuzzer in generating diversified inputs with its mutational algorithm;
3) The ability of a fuzzer to reach program points exploring intermediate inputs that are not an improvement in terms of coverage [5].

These insights can drive the user in the choice of the best technique to use for the selected program under test (PUT).

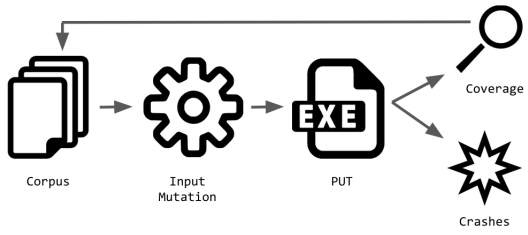
2. Background

The simplest description of Feedback-driven Fuzzing is an algorithm that provides apparently random data to a computer program and then it watches for crashes or unexpected states and also saves the generated input for later processing if they covers interesting new states that are used as feedback [6].

Tipically, the program states used as feedback are the edges in the program Control Flow Graph [7] in what is the so-called *Coverage-guided Fuzzing* (CGF).

The inputs are mutations of previously saved inputs in the fuzzing loop in Mutational Fuzzing or generated from scratch from a model in Generational Fuzzing.

Our baseline, AFL++ [8], a widely used fuzzer in the recent times, is a Mutational Coverage Guided Fuzzer.



State of the art Coverage-Guided Fuzzers encode the approximate executed path in a representation that is easy and fast to process. AFL++ uses a vector of 65536 entries, the *hitcounts* vector.

Each coordinate is associated with an edge and each value represents how many times the edge is executed modulo 256.

When a values greater than the previous one is registered in this vector, the fuzzer consider the input interesting and saves it.

Some extensions of CGF save also intermediate inputs that are a superset of the coverage reported in the hitcounts vector, like [9] [10] [5].

In general, when an input is saved, we can associate it to the testcase that generated it by mutation, the *parent* testcase. In this way, is easy to construct a tree of generated inputs that represents the progress of the hill climbing algorithm of the fuzzer, the *levels* tree.

Looking for parents in this tree, we can evaluate if the intermediate inputs generated by the specific technique is effective.

3. Proposal

We propose a time-based visualization of the progress of different techniques based on AFL++.

The first view will be a *navigable levels tree*. The user can select the technique in a scrollbar and view the associated tree.

Each node of the tree can be selected to view if it can be considered an interesting input for another technique. This will help in understanding the insight 3, were a tecnhique rech a program points thanks to an intermediate input that another technqiue cannot see as interesting.

There will be also a *scatterplot representing the inputs* in terms of covered state. We collect the hitcounts vector of each input, reduce the dimensionality using PCA and then apply t-SNE on the vector with reduced dimensions to preserve outliners in these most important dimensions.

The color of each point will be related to the technique that generated the input.

In this scatterplot, the user can manually select a cluster of points that represents inputs with related paths.

Of course it can choose to consider only a subset of the techniques and hide that points from the view.

Doing that, will help to cope the insight number 1 looking at the density of the cluster for each different technique.

When doing this, the user can then choose to highlight in the scatterplot the inputs that are parents for the points in the cluster.

A technque that is able to generate inputs from other that are almost completely unrelated have a better mutational algorithm than a technqiue that generate inputs similar to the parents, this helps the analyst with insight number 2.

The tree an the scatterplot are synchronized. When a cluster is selected, the corresponding inputs are highlighted in the tree.

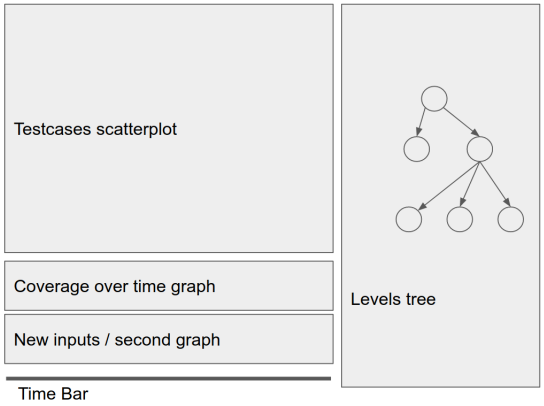
All these points can be hidden using a *time bar*. Selecting a specific lapse of time, only inputs founds by the fuzzer before that time are considered.

The other two views, related to the evolution in time, are the *coverage over time graph* and the *new interesting inputs per seconds graph*.

When a testcase is selected (both in the scatterplot or in the tree) a line appears in these graphs highlighing the time in which the testcase was discovered.

Coverage over time shows when an input that is important in terms of new coverage is found and so suggest the user where it should put the time bar for a targeted analysis.

Inputs per seconds is related with the levels tree. If a fuzzer generates too many intermediate testcases it can saturate the evolution of the alogrithm, so if we select an intermediate input in the levels tree and see an peak in this graph likely the technique suffers from path explosion.



In order to give an idea about dimensionality, we pick pcre2 version 183 (SVN) as example PUT. We compiled it with the AFL++ compiler wrapper and the addition of AddressSanitizer and UndefinedBehaviourSanitizer. After 12 hours of fuzzing, we got 14853 generated inputs in the fuzzer queue.

As said before, the hitcounts vector has 65536 dimensions, so in this simple case we have 973406208 elements.

References

[1] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[2] S. Poeplau and A. Francillon, "Systematic comparison of symbolic execution systems: Intermediate representation and its generation," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 163176. [Online]. Available: https://doi.org/10.1145/3359789.3359796

[3] 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. 21232138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[4] S. sterlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *USENIX Security*, Aug. 2020. [Online]. Available: Paper=https://download.vusec.net/papers/parmesan_sec20.pdfCode=https://github.com/vusec/parmesan

[5] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[6] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," https://www.fuzzingbook.org/, 2019, [Online; accessed 10-Sep-2019].

[7] K. Cooper and L. Torczon, *Engineering a Compiler: International Student Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[9] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[10] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *IEEE Symposium on Security and Privacy (Oakland)*, 2020.