FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques

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

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

Abstract—**Fuzz Testing techniques are the state of the art in software testing for security issues nowadays. Their great effectiveness attracted the attention of researchers and hackers and involved them in developing a lot of new techniques to improve Fuzz Testing. The evaluation and the cross-comparison of these techniques is an almost open problem. In this paper, we propose a human-driven approach to this problem based on information visualization. We developed a prototype upon the AFL++ fuzzing framework, FUZZSPLORE, that an analyst can use to get useful insights about different fuzzing configurations applied to a specific target in order to choose or tune the best technique during a fuzzing campaign.**

1. Introduction

Fuzz Testing or *Fuzzing* 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 academia 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 or by specific target-dependant actions taken by the fuzzer 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 immediate basic comparison between two or more techniques, an analyst has to just see which technique reaches more coverage in less time but does not reveal the properties of a fuzzer regards specific types of program states.

For instance, a technique can be better than another in exploring some types of program states and at 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].

The problem of the evaluation of fuzzing techniques is important not only when the aim is to generally states which fuzzer is best, but also when an analyst wants to select the best fuzzers for a single target. It is common that fuzzers that are considered generally better than others on some targets perform worst than the others [5].

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

The main insight that a 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 [6].

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

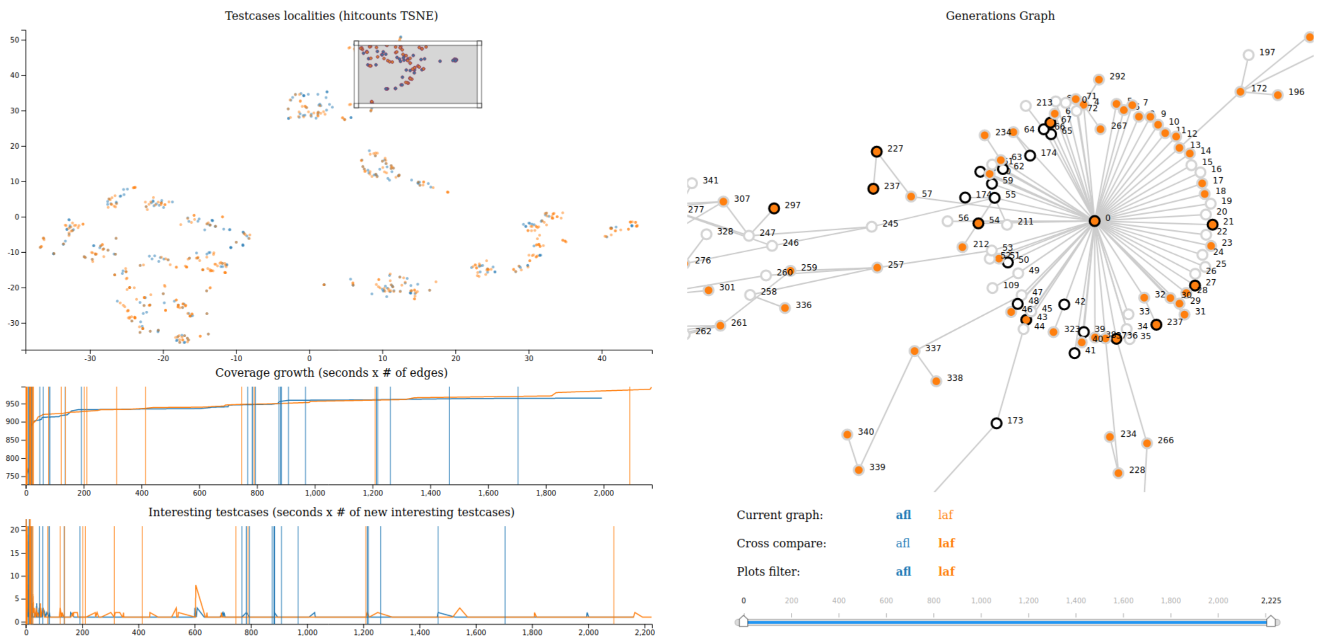
2. Background

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 cover interesting new states in terms of the chosen feedback [7].

Typically, the property of the program used as feedback is the set of the edges in the program Control Flow Graph [8] 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 (Figure 2) or generated from scratch from a model in Generational Fuzzing.

We base our implementation on AFL++ [5], a widely used fuzzer in recent times, that is a Mutational Coverage Guided Fuzzer.



Testcases localities (hitcounts TSNE)
Coverage growth (seconds x # of edges)
Interesting testcases (seconds x # of new interesting testcases)
Generations Graph
Current graph: afl laf
Cross compare: afl laf
Plots filter: afl laf

Figure 1: Complete view of the FuzzSplore visual panel.

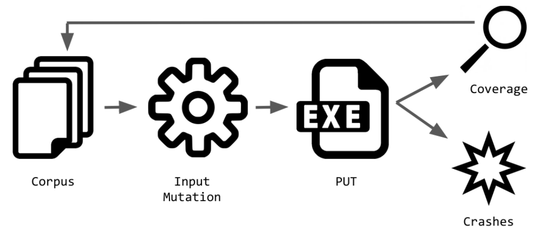


Figure 2: Basic representation of the Mutational Coverage Guided Fuzzing algorithm

State of the art Coverage-Guided Fuzzers encodes the approximate executed path in a representation that is easy and fast to process. AFL++ uses a vector of 65536 entries by default, 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 value greater than the previous one is registered in this vector, the fuzzer considers 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] [6].

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

3. Methodology

A *fuzzing campaign* is the process of running one or more fuzzers for a long period of time or even continuously like in OSS-Fuzz [11].

Security researchers typically start fuzzing using naive configurations and off-the-shelf fuzzers, then, meanwhile, the campaign runs, observe the evolution and tune the fuzzers.

Our proposed approach aims to insert in the observation-tuning feedback loop a visual component to help the researcher better understand insights about the fuzzers testing a particular target.

The data processed by *FuzzSplore* comes from the execution of the corpus of testcases that each fuzzer saved so far. The execution is instrumented and various properties are observed.

Then we visualize these collected properties and the user can relate them to better understand what is going. After that, the user can choose to drop some fuzzers if less effective and assign more resources (typically CPUs) to the most effective fuzzers or tune each individual fuzzer.

The fuzzing campaign can then continue. When it saturates, the analyst can collect insights using our tool and restart the visual analytics feedback-loop.

Saturation of fuzzers, when no more additional state is explored or the number of states explodes, is a problem that was rarely addressed in academic literature but that affects each type of Feedback-driven Fuzzer [12], and a tool that can guide towards the selection of techniques that avoid saturation can help a lot the campaign.

3.1. Data Retrieval

We denote each fuzzer F_i where i is the index that identifies it. With PUT_i we denote the version of the PUT preprocessed and instrumented in order to be used by F_i. PUT_e is the version of the PUT that logs the edge coverage using the hitcounts vector. It has to be provided independently if it is used or not by some fuzzer F_i. With $T_i(t)$ we denote the set of the saved testcases, the queue, by F_i until time t (seconds).

Given t as the time chosen by the user to observe the progress of the fuzzers, the Algorithm 1 computes the following sets:

- the set C of all the functions $C_i\colon Time \longrightarrow NumEdges$ that relates, for the fuzzer F_i, a time unit to the number of discovered edges so far;
- the set I of all the functions $I_i\colon Testcase \longrightarrow \{F_j, ...\}$ that associates, for the fuzzer F_i, each testcase in $T_i(t)$ to the set of fuzzers that consider the testcase as interesting;
- the set X of the sets X_i, that maintains, for each fuzzer, the hitcounts vectors associated with the execution of each testcase in $T_i(t)$;

Algorithm 1: Compute C, I, and X

```
for $F_i$ in $Fuzzers$ do
    $V_{acc} \leftarrow (0_0 ... 0_{65536})$
    for $T$ in $T_i(t)$ do
        $V \leftarrow Execute(PUT_e, T)$
        $X_i \leftarrow X_i \cup \{V\}$
        $V, IsInteresting \leftarrow$
          $MergeCoverage(V_{acc}, V)$
        if $IsInteresting$ then
            $C_i(Time(T)) \leftarrow CountNotZeros(V_i)$
    for $F_j$ in $Fuzzers \setminus F_i$ do
        $V_{acc} \leftarrow (0_0 ... 0_{65536})$
        for $T$ in $T_i(t)$ do
            $V \leftarrow Execute(PUT_j, T)$
            $V_{acc}, IsInteresting \leftarrow$
              $MergeCoverage(V_{acc}, V)$
            if $IsInteresting$ then
                $I_i(T) \leftarrow I_i(T) \cup \{F_j\}$
return $C, I, X$
```

The next item that has to be retrieved, in addition to C, I and X, is the set G of all the graphs G_i that describes the evolution of each $T_i(t)$, the levels graph introduced in Sec. 2.

We assume that each fuzzer encodes the information about the parent testcases into the metadata of each testcase. In this way, it is trivial to construct the graph just by reading all the metadata in $T_i(t)$.

3.2. Visualization

We visualize the computed data C, I, X, G, and some other properties that can be directly collected in four different views.

You can see these views with some example data in the screenshot of our implementation, in Figure 1.

A time bar is used to select $t' \in [0, t]$ to ignore data outside the selected time range and, for instance, visualize the data related to the queue $T_I(t')$ without the need to run again Algorithm 1.

3.2.1. Testcases Scatterplot. Each X_i is a matric of $|T_i(t)|$ rows in which each row is a vector of 65536 entries.

These raw numbers are raw to visualize. To handle this problem, we reduce the dimensionality of each vector $X_{i,j}$ from 65536 to 2, in order to be easily visualized in a scatterplot.

To do that, we chosen an algorithm that optimizes the conservation of local distances after the dimensionality reduction, *t-SNE* [13]. The nature of this algorithm is random, it needs to process X entirely in order to get new vectors that are meaningfully comparable.

We experimentally observed on a test dataset that a perplexity of 30 is good enough.

The user can select groups of nodes interactively to highlight properties in the other visualizations.

3.2.2. Coverage Growth Plot. C can be visualized simply using a line plot with the X axis representing the domain, the time, and the Y axis the number of edges.

When a testcase is selected in the scatterplot or in the generations graph a vertical line appears at position x where x is the time in which the testcase was discovered.

3.2.3. Interesting Testcases Plot. This plot is used to visualize the evolution of the fuzzing algorithm in finding new testcases. The X axis represents time in seconds, the Y axis the number of new interesting testcases saved by the fuzzer in that second. This information is directly contained in $T_i(t)$.

Here too, when a testcase is selected in the scatterplot or in the generations graph a vertical line appears at position x where x is the time in which the testcase was discovered.

3.2.4. Generations Graph. We visualize each Generations Graph G_i combined with I. Given a fuzzer F_j from the user, we highlight in graph G_i each node associated with each testcase T if $F_j \in I_i(T)$. In this way, the user can know if the evolution of $T_i(t)$ associated with the fuzzer F_i is compatible with the selected F_j.

When a testcase is selected in the scatterplot, the border of the corresponding node in the graph is highlighted. The user can select additional nodes or deselect nodes selected from the scatterplot. The scatterplot selection is synchronized in both ways with the graph.

3.3. Analyst Feedback

The insights that an analyst can retrieve in order to choose or tune the fuzzers using the visualization are, but not limited to, the following:

- Looking just at the scatterplot, the user can select a subset of fuzzers that explore different program points if the points related to each fuzzer in the graphs are clustered;
- Looking at the scatterplot and the coverage graph, the user can select a cluster of testcases that are similar and see the ability of a fuzzer in generating similar testcases in a small range of time. A fuzzer that discovers few points at a time and have them distributed for all the X axis of the coverage plot should be deprioritized;
- Looking at the coverage graph, when there is a huge increment of the number of edges, the user can see if an outlier in the scatterplot was generated. This allows to isolate interesting testcases that improves a lot the coverage;
- Selecting testcases in the graph, the user can see if the testcases are similar in the scatterplot in order to understand the ability of the mutator to generate similar or different derived inputs;
- Selecting testcases in the graph and a fuzzer to cross-compare, the user can know if the coverage metric of the other fuzzer is sensitive enough to cover the selected testcases.

With this information, the security researcher should be able to choose and tune the set of fuzzers to avoid the saturation of the fuzzing campaign. This methodology is a first step towards a fuzzers debugger that is highly demanded by the security research community.

4. Implementation

We created an HTML page comprised of 4 views and a filtering panel and all the components were created using the D3.JS library.

4.1. Testcases Scatterplot

The scatterplot (Fig. 3) has, as both axes, a linear scale where the points are color-coded to represent a category to help the analyst distinguish the similarity in the clusters highlighted and the presence of outliers.

By brushing over the scatterplot a routine is called to update the other 3 views with the highlighted elements by selecting the corresponding nodes in the Generation graph and inserting lines in both plots.

The user can also zoom in and out and both axes are scaled appropriately.

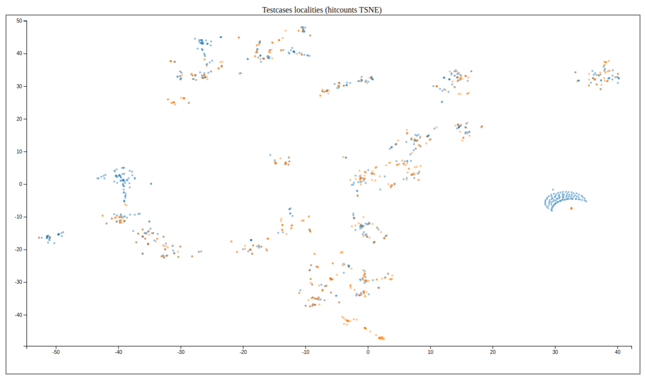


Figure 3: Testcases scatterplot

4.2. Coverage Growth and Interesting Testcases Plots

The Coverage graph (Fig. 4) plots the growth over time of the number of covered edges, the Interesting Testcase graph (Fig. 5) plots the number of new interesting test cases over time instead, for both the bottom axis is implemented as a linear scale, for the first graph the left axis is implemented as a logarithmic scale, for the latter a linear scale is used instead.

When data is selected on the scatterplot or graph vertical lines appear in both plots at the corresponding time having the stroke color matching the fuzzing technique.

We also implemented a pan and zoom functionality that keeps the lowest value pinned at the bottom.

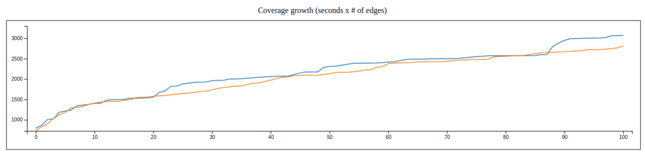


Figure 4: Coverage growth plot

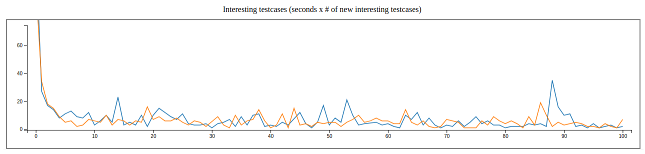


Figure 5: Interesting Testcases plot

4.3. Generation Graph

The Generation Graph (Fig. 6) is created as a hierarchical layout where each data point's value is displayed as a node label.

The user can zoom as well as pan over the entire view to have a better understanding of the data and when a node is selected in the other a routine is called to highlights the corresponding points in the scatterplot and insert lines in the other plots. A mouseover on node lowlights all the nodes except the hovered nodes and their neighbor nodes and edges.

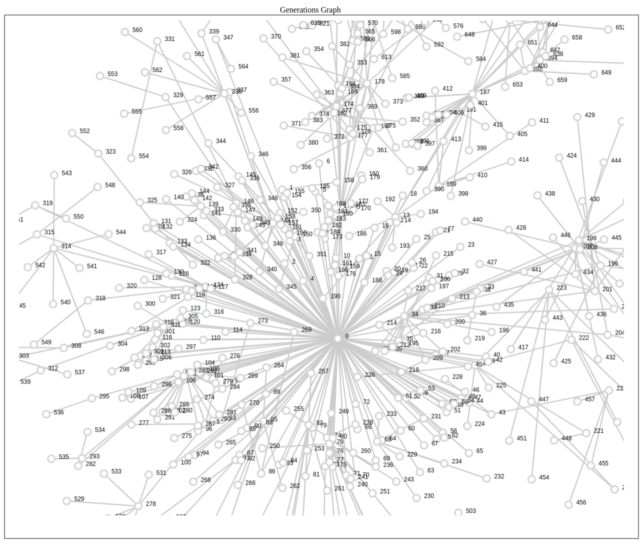


Figure 6: Generation graph

4.4. Filtering Panel

The user can filter the data shown an all the 4 views by time, with a range slider (Fig. 7) located at the bottom right of the page, and by category, by clicking on the category names directly on top of the slider, the filtering works by updating the existing graphs without redrawing.

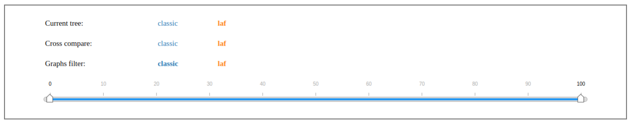


Figure 7: Filtering panel

5. Concluding Remarks

FUZZSPLORE brings a useful visualization-based method to retrieve insights from running fuzzers in a campaign.

It defines a long-term visual analytics feedback loop applied to fuzzing with a set of data retrieval and visualization techniques that can be easily extended in future works.

The information that a security researcher can collect using our approach can help in understanding the problem of saturation in fuzzing campaigns, a widely spread problem that is rarely addressed in academic literature.

We share FUZZSPLORE as Free and Open Source Software at https://github.com/andreafioraldi/FuzzSplore.

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. 163–176. [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. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[4] S. Österlund, 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] 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. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[6] 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

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

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

[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.

[11] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," in *USENIX Security Symposium*, 2017.

[12] A. Groce and J. Regehr, "The Saturation Effect in Fuzzing," https://blog.regehr.org/archives/1796.

[13] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.