CS 7250 - Information Visualization Final Project
Visualizing Program Coverage

Giorgio Severi, Andrew Fasano, Talha Paracha

Abstract

1 Introduction

A surprisingly difficult component of finding bugs in computer programs is to test 100% of the code in a program. Running a program on common types of inputs often fails to reveal bugs because, if these commonly broke, the developers would notice and fix the bugs. Instead, bugs often lie in uncommonly evaluated parts of a program. With this in mind, bug-finders typically try to find all these rarely executed sections to test if they contain bugs. The metric used to describe this is known as "coverage," a percentage of the program that a bug-finder is able to execute using many inputs. Bug-finding systems try to maximize this coverage in order to find as many bugs as possible.

A computer program can be represented by a large network where an execution of the program is a path through the network across different nodes. As time goes on, bug-finding systems are trying to reach all the different nodes in this network. We plan to visualize how these systems explore this network over time to enable data-exploration. By visually representing this data, we hope to identify patterns where different types of bug-finding systems fail to cover similar parts of programs.

1.1 Dataset

We will use the inputs submitted by competitors in the "Rode0day" bug-finding competition[1] and/or data generated by bug-finding systems such as AFL that we run. For each input generated by a bug-finding system, we will extract a list of basic blocks covered (the nodes covered by the input) and use these for our visualization. We have already collected the list of basic blocks covered by each of the 2205 inputs submitted by 20 teams participating in Rode0day.

[1] `rode0day.mit.edu/archive`

2 Interview

Due to the research oriented nature of the data we will be working on, the number of experts in the area is very limited. We will interview Josh Bundt, a graduate student in Khoury currently researching techniques to improving bug-finding with fuzz testing systems. The Interview is reported in the following paragraph.

2.1 Introduction

Can you tell us a little about your research interests?

Interested in system security, program security, software security; particularly interested in fuzzing.

How long have you been working the field of program analysis?

A couple of years.

What kind of tools do you use during your work?

For fuzzing; AFL+helper tools, GCOV, LCOV, Dynamario (execution traces), IDA Pro Other than that for research purposes; Ansible (IT automation), docker (reproducibility), Git

How often do you have access to the source code of a program you're analyzing?

60% black box, 40% source code.

What are the implications on your work if you do not have access to the source code of a program?

Having source makes it more work because you have more things to try. Not having source limits the tools.

2.2 Visualizations

Do you use any of the common visualization types during your work? (bar chart, line graphs, pie charts etc)

Largely text-based, AFL produces charts with GNUPlot; scatter plots to show progress over time etc, coverage tools produce a summary with highlighted color. Lighthouse with ADA[2] if you don't have access to code.

Do you use any interactive visualization types during your work?

IDA is interactive, coverage reports in HTML are somewhat interactive (taking user from code to function source)

[2] `https://github.com/gaasedelen/lighthouse`

How pleased are you about the aesthetics of the visualizations generated by the tools you use. Do you wish them to be better? Do you think it matters in helping you be more productive?

Lighthouse is visually appealing, other tools are just useful

Do you encounter situations where you have to share the visualizations with people outside of your area? Do you want your visualizations to cater to general-audience?

Occasionally with people in specialized area, rarely with outside CS/normal people.

Do you prefer to use GUI-based tools for generating visualizations or code-based?

Prefer to code-based because easier to use.

2.3 Fuzzing

How do you measure success when fuzzing a program?

Can be difficult, most fuzzers can only tell whether the program crashed, in fuzzing research figuring out useful instances (from all crashes) can be really difficult. Coverage is kind-of measure of success.

Is there any kind of visual aid you could use here to better understand when you're finding new parts of a program?

CGC interface: See where two execution of programs diverted, Might be easier to bucket a thousand crashes into similar causes (etc.), currently not a solved problem.

If you're getting feedback suggesting that things aren't going as well as expected, do you find a way to fix the problem, move on to another target, or just ignore the feedback altogether?

If it shows that you aren't getting good coverage, then maybe you set things up wrong and should try again. Other than that, maybe just ignore it

Is there any particular limitation in the UX of your tools that you'd like to change?

AFL has a text-based interface for many results; some are stored in a file, others available only by looking at them visually, trade-off between performance vs saving extra information. AFL has a queue of inputs with different paths for different coverage (but cannot validate that), nice to have a better representation of coverage. Afl has three classes of "paths; this many pending; to convey coverage information. It is not based on actual program coverage.

Would your productivity benefit from being able to observe the code coverage in real time during a fuzzing campaign?

During research it could give you a better idea of "stopping point", should be interested in knowing what part of code you wanted to cover and everything that it calls,

Would your productivity benefit from having an interactive visualization showing coverage plus the ability to guide a fuzzer down a specific path in real time?

Yes. Another aspect of AFL is that it picks favored inputs but don't know what algorithm it uses to Not transparency and doesn't let user suggest things to favor (or stop)

What is the basic unit of measure (code block, line of code, function) that you use when fuzzing?

Most common; basic blocks or looking edges of the control flow graph More granularity; machine instructions, line of code

3 Task Analysis

4 Implementation

References