Invariant-based AFL Fuzz testing

Wen-Chi Hung
Department of Computer Science
University of California, Davis
wenhung@ucdavis.edu

Xinyi Zhang
Department of Computer Science
University of California, Davis
xykzhang@ucdavis.edu

Yiran Wang
Department of Computer Science
University of California, Davis
wyrwang@ucdavis.edu

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. Some fuzzers (like AFL and LLVM?s libfuzzer) have instrumentation compiled inside the tested software so the fuzzer can see how the program behaves internally. While fuzzing tools are generally aimed to explore and test undefined areas that may trigger bugs, how to get those test cases for an undefined area is the core problem.

American fuzzy lop(AFL) is a well-known technique that has successfully detect significant software bugs in dozens of major free software projects. The strategy for AFL is to mutate a user-defined test case by applying various modifications to initial user input. The core procedures of AFL: 1, take next input file from the queue 2, mutate the file 3, record unique mutate output. The iterations of these three steps will be repeated until new internal states in the targeted binary been triggered. AFL performs well to find some bugs, but it may take a very long time to trigger those interesting results. The mutation of test cases can be forever in order to be explicit in this exploring process. Some techniques, thus, are used to prevent redundant and repeated works by choosing different exploration paths as much as possible. The AFL uses coverage test as the standard to distinguish test cases. In another word, starts from that initial stage, it automatically discovers clean, interesting test cases by coverage test.

2 Motivation

While an important step of fuzzers, like AFL, is to repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies, it?s not trivial to recognize those ?clean and interesting? test cases to do the real testing. If the test inputs generated by mutating cannot be effectively recognized as a repetition of an existing one, redundant work will be done again. The goal is to have an effective method to cover more program paths and trigger unexpected behavior. While the coverage test is used in the current version of AFL, we want to explore whether there?s a better way, such as using ?invariants?(relatively ?invariants?) as criteria. If we could reduce more repetition, the efficiency of AFL can be improved.

ECS 289C, Programming Languages and Compilers, Davis, CA USA 2017.

3 Solution

In this project, we would like to apply the comparison of ?invariants? in programs instead of just the criteria based on the coverage in AFL. The ?invariants? here is not the true invariants for the program, but relatively unchanged variants compared with several inputs. To find the ?invariants? in the program, we plan to use Daikon. Daikon is an implementation of dynamic detection of likely invariants. During the detection process, the Daikon invariant detector reports likely program invariants. If the ?invariants? detected by Daikon are different from the ?invariants? from previous inputs, we save the mutation for future variations. There are three plans to integrate the ?invariants? detection with the original AFL and we plan to compare the results of all those and the original AFL.

1. Replace the coverage test.
2. Apply invariants test to the results from coverage test.
3. Combine the results from invariant and coverage test.

In AFL, the mutated test cases are pushed into a queue if they are considered ?interesting?. We plan to replace that method that detects only the coverage to a system call to another application(Daikon) that return a pool of ?interesting? cases back to the AFL. We plan to modify the AFL to take new test cases in the three plans we mentioned before.

4 Comparison

For the coverage-based fuzzer, if a mutation triggers execution of a previously-uncovered path in the code under test, then that mutation is saved to the input pool for future variations. We want to add invariants test to the fuzzer. The fuzzer generates random mutations based on the sample inputs in the current corpus. Besides the coverage tests in original AFL, we use ?invariants? to consider whether this test case should be use or not less. , if certain test cases generates a new a set of ?invariants? that differ from previous from Daikon, we will continue the AFL to test the program using this new mutation and take it as the seed to generate more test cases.

5 Related Work

Fuzz testing. Fuzz testing is an automated software testing technique. Depending on whether the fuzzers are aware of program structure, fuzzers could be categorized as white-, grey-, or black-box fuzzers.

Symbolic execution based white-box fuzzers like Klee[1] or Katch[2] use symbolic program analysis and constraint solving to accurately direct the search in test generation as and when required.

American Fuzzy lop (AFL)[3] is a coverage-based fuzzers. Unlike symbolic execution, AFL has the ability to generate lots of test cases. Its attempt to cover more program paths using mutations of seed inputs without incurring any overhead of program analysis. Some improvements have been introduced based on AFL. In the paper of AFL Fast[4], they present the structure of coverage-based Greybox Fuzzing (CGF). The recent work AFLGo[5] is focusing on directed grey box fuzzing. It can achieve higher performance if there is a specific target to be tested. However, in our experiment, we apply the original edition of AFL because the possible application can be more general.

Since the AFL conduct random mutation to generate test cases, straying to the wrong direction can cause huge overhead. Combining the AFL to invariant-based testing can be a solution. Because the huge amount of test cases generated by mutation, the invariant-best testing tool can utilize those test cases to check whether they are in the range of current invariants. On one hand, those test cases remained can be applied on the invariant -best testing tool to produce more precise invariant; on the other hand, since the invariant-best testing tool delete some cases generated by AFL, the AFL can focus on those more promising routes.

Discovering Program Invariants . Daikon[7] can conduct dynamic inference of invariants for applications in software evolution. Dynamically inferred invariants also could be used in many situations that declared or statically inferred invariants can and, in some cases, the application of dynamic ones may be more effective. Invariants have many uses in computer science, and is considered as an indicator of triggering new running status of the program.Since the invariants generated by Daikon represent the current status of the program with input test cases, test cases which are out of the range of invariants are highly possible to discover a new path for the target program. It is thus a possible method to be used to eliminate useless data from the list of mutation test inputs from AFL. In addition, Daikon has the compatibility in C language which is ideal for integrating with AFL. Other tools that we considered such as Deduce[8], are simpler than Daikon, but doesn?t support C program.

5.1 Tools

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

A Appendix

Text of appendix ...