CS4110: Software Reversing Lab 1

Reverse Engineering Group 15
Mark van Beusekom (4029127)
Richard Luyckx (4324765)
Wendo Sabée (4023978)

March 20, 2017

1 Introduction

Automated Testing is an important part of software engineering, but one often neglected in practice. This despite that the advantages are plentiful and clear. Regression testing for instance prevents new bugs being introduced after an implemented fix. One of the reasons for this is because programmers have a tendency to consider writing tests as being secondary to writing production code, and as such do not invest the proper amount of time into writing tests. Poor code coverage is another challenge within the field of software testing, where the code is partially tested but not enough to catch the more difficult to find bugs. This report documents a number of automated techniques designed to discover bugs in production code. Chapter 2 describes fuzzing: a method to generate smart inputs in order to find bugs in the code. Concolic testing is explained in chapter 3, a technique that explores unique paths in the source code in order to find bugs. Following that, chapter 4 discusses tools that can be used to facilitate fuzzing and concolic testing. These are finally put to test in chapters 5 and 6

2 Fuzzing

Fuzzing is a technique allows quality assurance teams to test large numbers of boundary cases while implementing these with conventional unit tests would be extremely time consuming. [3] The concept behind fuzzing is to make a given piece of software display aberrant behavior by generating (semi-)random inputs, rather than having humans come up with more specific test cases. This is generally a time-consuming job and as such it may be more efficient to have a computer do it. The traditional form is a technique called black-box fuzzing which runs the random inputs on a given binary file without knowledge of the source code [2]. Black-box fuzzing does have significant limitation, as we can show with snippet 1.

Snippet 1 : Low chance of choosing erroneous value

```
int prog(int x){
   int y = x+3;
   if(y == 13) \{
      abort(); /* error */
   }
   return y;\\
}
```

Assuming a 32 bit integer is used, a completely random fuzzer has a 1 in 2^{32} chance to trigger the bug. This intuitively explains why black-box fuzzing typically suffer from achieving low code-coverage. [2]

Thanks to recent advances in systematic dynamic test generation white-box fuzzing has become more popular. This technique analyzes the source code of the program in order to generate sensible inputs based on the branches in the code. For instance in snippet 1, it would discover that `x+3 != 13` and then solve the input to `10`, triggering the bug. Using these analytical methods rather than the random methods as used by black-box fuzzing, we can reach a significantly higher code coverage.

3 Concolic execution

Concolic testing (which stands for *conc*rete and symb*olic* testing) is another technique that allows quality assurance teams to generate large amounts of tests in order to ensure quality code. it builds upon the concepts from white-box fuzzing in that it uses the same branch constraints. A key difference between concolic testing and fuzzing lies in path exploration. While a fuzzer merely attempts to find new interesting inputs, a concolic tester analyzes the program trace in order to find new interesting paths and derives inputs for them. It does this by running a program with an arbitrary input, and generating a program trace. Upon analysis of this trace it generates a set of constraints for each branch. The algorithm then tries to come up with the next unexplored path using a SAT solver, generates the proper inputs for said path and repeats the process for it.

Snippet 2 : Concolic execution example

```
void f(int x, int y) {
   int z = 2*y;
   if (x == 100000) {
      if (x < z) {
         assert(0); /* error */
      }
   }
}
```

If concolic testing is performed on code snippet 2, it will start with entering arbitrary values for X and Y, leading most likely to the `if (x == 100000)` branch being false, thus no further code is reached. On the next iteration, it'll attempt to make the last branch found in the code true in order to discover a new path. using the constraint `x=100000` it will discover the new `if (x < z)` branch, where

`z` is an arbitrary value based on the value of `y`. If the branch is true, it will find the error and continue searching by trying to make the constraint `x < 2*y` false. Likewise, if the branch was false, it would continue searching by making the constraint `x < 2*y` true, and find the error. As there are no more branches left to explore, the algorithm completes.

4 Tool description

4.1 AFL

American Fuzzy Lop (AFL) is a tool to execute previously discussed fuzzing techniques. The tool is described as a brute-force fuzzer coupled with a genetic algorithm [4]. For detecting subtle changes in the program flow it uses a modified form of edge coverage.

In simplified form, the algorithm AFL uses can be summed up as:

1. Load user-supplied initial test cases into the test queue.
2. Take the next input file from the queue.
3. Try to reduce the test case size without altering the measured execution states of the program.
4. Mutate test files.
5. If the mutation triggers a new transition, add the mutation to the test queue.
6. Restart from 2.

The test cases it finds are saved, for example for future use of further testing, and can be run on the software it was fuzzing to reproduce and trace the fault.

The use of combined methods for fuzzing provides in an efficient guided fuzzing tool which greatly outperforms blind, random fuzzing. This is shown in their technical "whitepaper" found on the website, where they show the difference of blind fuzzing versus the AFL model, both executed on the GNU patch 2.7.3 and seeded with a dummy text file. The AFM model reached about ten times more blocks, and also ten times more edges.

4.2 KLEE

KLEE is a tool that can automatically generate high coverage tests[1]. It does so by using symbolic execution and a SMT solver to find execution paths that generate errors.

It works by first modifying the program by marking one or more inputs as symbolic with the `klee_make_symbolic()` function. These inputs will not be evaluated with actual values during execution, but rather with symbolic values. KLEE works with LLVM bitcode files, which is more or less an intermediate format which contains a platform independent assembly. The program is compiled to these bitcode files.

It then uses symbolic execution to find a list of execution paths, by default using a *Random Path Selection* strategy. It keeps a state for each of the steps

	# of inputs	LOC
Problem 5	5	9136
Problem 10	5	1617
Problem 13	10	118323
Problem 15	20	155758
Problem 18	20	593994

Table 1: Overview of selected problems

in the execution path, and for each branch it find (for example, an if statement) it "forks" this state and explores both (or all) possibilities. For each of these execution paths, it generates a list of predicates and converts and optimizes these into a query that the SMT solver can understand. By default, it uses the STP solver but KLEE can also be configured to use the MetaSMT or Z3 solvers.

Using these solvers, KLEE finds a (list of) inputs that will lead to the execution of each of the found execution paths. Each of these inputs is a new test case and if the test case generates and error, the type of error and it's output will be logged.

5 RERS problems

For the experiments a selection was made from the collection of RERS reachability problems available. We have selected problem 5, 10, 13, 15 and 18. The selection was made based on the number of inputs the programs accept, and the number of lines of code (LOC) they have, to get a "test set" with a good range of size and input possibilities. Initially, the tools were ran with a duration of 5 minutes for problems 5, 10, 13 and 18. Problem 15 was added later to do a much longer run: approximately 60 minutes. Table 1 shows an overview with the selected problems.

6 Result analysis

The tools each provide with different outputs in which the results are presented. These will be explained in this chapter as well as some result differences between the tools.

6.1 AFL

AFL creates three output directories at the start of execution, in which the results are stored. The files in these directories are updated in real time during the execution of the program, and can be evaluated afterwards. The directories are called "queue", "crashes" and "hangs". The queue directory contains test cases for all unique execution paths and the starting test files provided by the user. Test cases that trigger the error in the program are saved in the crashes directory to easily identify test cases which trigger the crashes. The hangs directory contains test cases that caused the program execution to time out. By default, afl uses a timeout period which is quite agressive, which might give some noise in execution due to normal latency spikes for example.

The files contain what the input to the program was during execution, providing the user with information about the trace and the desired inputs to reproduce the crash.

6.2 KLEE

For each execution of KLEE, a new output directory is created. This directory contains a few different type of files. All `kquery` files that contain the queries that are used by the SMT solver can also optionally be outputted as `smt2` files.

6.2.1 Global files

These files contain information about the whole execution progress of KLEE. It contains information about how KLEE was executed (`info`), information about messages emitted by KLEE (`messages.txt` and `warnings.txt`), a humanreadable bitcode version of the program (`assembly.ll`) and statistics about the KLEE execution that can be instpected with the `klee-stats` tool (`run.stats` and `run.istats`).

KLEE can also record all queries it generates before optimization and passing it to the solver (in the `all-queries.kquery` file), or all queries after optimization that are actually passed to the solver (in the `solver-queries.kquery` file).

6.2.2 Per-path files

For each execution path that KLEE finds, it will generate several files. These files are named `test<n>.ktest` where `<n>` is the test number. These files can be inspected by running the `ktest-tool` on them. It will show the input of all the symbolic variables that lead to the execution of this path.

Whenever the execution path creates an error, the output of the program will be logged in the `test<n>.<error>.err` file that will be created, where `<error>` is the type of error that occurred. These files were used to determine which and how many of the error conditions were found in the execution of the RERS problems.

The query that was given the the solver that lead to this execution path is also logged in the `test<n>.kquery` (or optionally) `test<n>.cvc` and `test<n>.smt2` files. These all contain the same information but in a different format. Using the `kleaver` tool, the `kquery` files can be solved.

6.3 Findings

In table 2 we present an overview of the amount of problems found by the testing tools compared to the total amount of reachable problems in the code, the latter was gathered from the RERS challenge website. The full results can be found in table 3.

Analyzing which problems were found was done by filtering all unique error codes and comparing the outputs of AFL and KLEE. In all cases but one, KLEE found all problems that AFL also found or more. The exception to this is Problem 5 (15 min). For this problem, KLEE found 24 problems while AFL found the same 24 problems and one more. Since this additional find by AFL is a very big problem with 70 inputs, we could say that AFL got 'lucky'.

	AFL	KLEE	Total present
Problem 5 (5 mins)	24 (77%)	24 (77%)	31
Problem 5 (15 mins)	25 (81%)	24 (77%)	31
Problem 10 (5 mins)	45 (100%)	45 (100%)	45
Problem 13 (5 mins)	13 (43%)	23 (77%)	30
Problem 15 (5 mins)	19 (59%)	26 (81%)	32
Problem 15 (60 mins)	23 (72%)	26 (81%)	32
Problem 18 (5 mins)	17 (53%)	24 (75%)	32

Table 2: Overview of results vs total present problems

	Problems found by		
Problem number	AFL	KLEE	Both
Problem 5 (5 min)			0, 4, 5, 6, 7, 13, 14, 18, 31, 35 40, 41, 42, 46, 52, 61, 62, 68, 69 76, 80, 86, 97, 98
Problem 5 (15 min)	92		0, 4, 5, 6, 7, 13, 14, 18, 31, 35, 40, 41, 42, 46, 52, 61, 62, 68, 69, 76, 80, 86, 97, 98
Problem 10 (5 min)			ALL
Problem 13 (5 min)		15, 21, 30, 33, 50, 53, 59, 81, 92, 96	6, 13, 18, 23, 25, 27, 40, 47, 78, 83, 87, 93, 95
Problem 15 (5 min)		15, 23, 26, 57, 65, 67, 72	3, 10, 16, 22, 40, 42, 45, 46, 49, 58, 61, 63, 75, 78, 82, 89, 91, 93, 95
Problem 15 (60 min)		15, 57, 67	3, 10, 16, 22, 40, 42, 45, 46, 49, 58, 61, 63, 75, 78, 82, 89, 91, 93, 95
Problem 18 (5 min)		3, 14, 39, 53, 70, 71, 76	0, 11, 17, 20, 24, 25, 38, 46, 48, 62, 78, 81, 85, 91, 93, 95, 99

Table 3: Results of all findings, lists what tool found what error code

In all other cases where the number of found problems differs between the two tools, KLEE always found the same problems that AFL found and more. Problem 13 really demonstrates that KLEE is a more efficient than AFL, because 8 of the 10 problems that it additionally found only have 3 or 4 inputs. If AFL had been given more time than 5 minutes, it would have likely found those too.

This difference in efficiency is again demonstrated with Problem 15. When both tools run for 5 minutes, KLEE finds 7 more problems than AFL. After 60 minutes however, the difference has shrunk to only 3 problems, two of them big with 11 inputs. For problem 18, KLEE finds 7 more problems than AFL in those 5 minutes, of which a majority of 5 problems were big (with between 11 and 22 characters) and only two were small (with 3 inputs)

Generally, we can conclude that AFL will eventually find most or all of the problems that KLEE will find (depending on the problem), given enough time. KLEE is however faster and more efficient in doing so.

References

[1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[2] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[3] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[4] Michal Zalewski. American fuzzy lop, 2007.