University of Twente

Software Testing and Reverse Engineering

Software Reversing Lab 1

Authors:
Joeri Kock
Æde Symen Hoekstra
Erik Kemp

March 21, 2017

1 Introduction

This is the report for the first lab assignment for Software Testing and Reverse Engineering. First we explain Fuzzing and Concolic (Concrete and Symbolic) execution in general with small examples. Then we will explain the functionality and implementation of AFL [1] and KLEE [2]. Afterwards, we will introduce the experiments we conducted with AFL and KLEE on some of the RERS [3] Reachability problems and analyse the results in detail.

2 Fuzzing

Fuzzing is a method for software testing involving sending data that is either invalid, unexpected or completely random to a computer program. The way this computer program handles the data is monitored, so one can find e.g. crashes, exceptions and memory leaks in the program. Fuzzing is a testing method that works best on computer programs that are only meant to receive structured and valid data, and exploit vulnerabilities that cause the program to crash, such as buffer overflow, cross-site scripting, denial of service attacks, format bugs and SQL injection.

Lets us consider an integer in a program, which stores the result of a user's choice between 3 questions. When the user picks one, the choice will be 0, 1 or 2. Which makes three practical cases. But what if we transmit 3, or 255? We can, because integers are stored a static size variable. If the default switch case hasn't been implemented securely, the program may crash and lead to security issues. A fuzzer can effectively identify these kind of vulnerabilities in a system.

3 Concolic Execution

Concolic execution is a mix between concrete and symbolic execution, with the purpose of making the process of symbolic execution more feasible.

Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions. The complication of symbolic execution is that, except for micro benchmarks, the cost of executing a program through all possible execution paths is exponentially large, thus prohibitive. However, if we provide the symbolic execution with concrete values, you can guide it through a specific execution path (without traversing all of them) and achieve the respective path condition. This is more feasible for larger and more complex computer programs and results in lower costs overall.

For an example, consider the following code:

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
        }
    }

int main() {
  x = read();
```

```
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input. Suppose we read x = 22 and y = 7. The concrete execution will now proceed to the 'else'-branch of the first if-statement, since x != z. At this point, the concolic execution can decide it wants to explore the 'true' branch instead. Hence, it will return values for x and y in order to satisfy the first condition. Suppose it returns x = 2 and y = 1.

In this case, concrete execution will reach the 'else' branch of the second if-statement. Again, concolic execution wants to explore the 'true' branch of this statement. Suppose it generates x = 30 and y = 15. If we run the program again, it will reach the ERROR state.

As we can see from this example, using concolic execution, we can use the information of the symbolic execution to obtain new inputs.

4 AFL and KLEE

In this paper, we compare our results between AFL and KLEE. Since AFL is a fuzzing tool and KLEE a symbolic execution tool, it's clear that they differ in many aspects. In this section, we will investigate the functionality of AFL and KLEE in more detail and analyse what their strengths and weaknesses are.

4.1 AFL

The American Fuzzy Lop is created and maintained by Michal Zalewski, a Polish white-hat security expert who is currently employed by Google. An introduction to AFL can be found and the readme-page of the website[4]. In his introduction to the challenges of guided fuzzing, Zalewski writes that fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software. In addition, he states that a fuzzing is also relatively shallow in its abilities to find certain code paths. He also mentions more sophisticated research subjects as concolic execution, symbolic execution and static analysis, but states that these tools currently do not offer a viable alternative to 'dumb' fuzzing techniques due to reliability and performance problems.

AFL needs two files as input: The program it will analyse and a base test case (or multiple test cases) from where it will create mutations. AFL works by repeatedly running and mutating the test cases. For the mutation process, AFL uses well-researched fuzzing strategies. Early in the process, these strategies are highly deterministic, with amongst others:

1. Sequential bit flips with varying lengths and stepovers
2. Sequential addition and subtraction of small integers
3. Sequential insertion of known interesting integers (for example: 0, 1, INT_MAX)

In a later stage, when there are multiple compact test cases, AFL starts to use more non-deterministic mutation strategies, such as:

1. Stacked bit flips
2. Insertions
3. Deletions
4. Arithmetics

5. Splicing of different test cases

The strength of AFL comes in part from the fact that it can reuse existing test cases. So if any other tool has already created test cases for the program of analysis, AFL can use these tests as input. In fuzzer terminology, blackbox fuzzers are totally unaware of the internal program structure. Whitebox fuzzers systematically increase code coverage through program analysis. Greybox fuzzers use instrumentation to get a little bit information on code analysis. AFL falls in the last category, by using lightweight instrumentation to inform itself about code coverage. This leads to a significant performance overhead, but leads to AFL being an extremely efficient tool to detect vulnerabilities. This is more thoroughly explained in Bohme et al. [5].

4.2 KLEE

KLEE is a testing tool that is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. Contrary to AFL, KLEE uses symbolic execution to automatically generate test inputs, making it more efficient than a regular fuzzing tester.

When testing code with KLEE, the inputs have to be made symbolic. KLEE has a function for this, which takes the address of the variable, its size, and a name. After this, a bitcode file is generated and one can run KLEE on this file. KLEE will then analyze the file, and inform the user about the number of paths in the program. Furthermore, KLEE has generated one test case for each path explored, which are all stored in the KLEE output directory. The test cases are the **.ktest** files. Using the **ktest-tool** (which is built in to KLEE), we can examine each file individually. In these files, we can see the value of the inputs for each path in the code. For example, if a code has three paths, we will see 3 values corresponding to each of the paths. Lastly, each of these paths can be replayed using KLEE as well by running the **ktest** files again. KLEE will replay the test case and the correct path, and give the output to the user.

As we can see, KLEE is very useful for larger and more complex programs. In comparison, AFL will test a program according to all possible execution paths, which is not feasible for larger programs due to its exponential scaling. KLEE, however, is able to detect all paths in the program individually and distinguish them, and generate a test case for each unique path in the code. While AFL is more thorough, KLEE is in a way 'smarter' than a fuzzing tester, and thus more feasible for more complex code.

5 Experiments

We ran both AFL and KLEE on Problem 13, 14 and 15. We used the tools to calculate the number of unique crashes of each problem, which are listed in the table below:

size	plain	arithmetic	data structures
medium	**Problem 13**	**Problem 14**	**Problem 15**
	AFL: 378	AFL: 237	AFL: 204
	KLEE: 28	KLEE: 27	KLEE: 6

From KLEE, we can also extract the following statistics for Problem 13:

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)	States	maxStates
klee-last	1280523277	8716.64	32.44	45.37	224940	8.44	0	33553

avgStates	Mem(MB)	maxMem(MB)	avgMem(MB)	Queries	AvgQC	Tcex(%)	Tfork(%)
25420.48	176.50	2095.35	1882.78	4302	96	2.70	5.02

From AFL, we have extracted the following relevant statistics for Problem 13:

elapsed_time : 02:25:17
elapsed_time(s) : 8717
cycles_done : 2
execs_done : 22877538
execs_per_sec : 589.36
paths_total : 3065
max_depth : 48
bitmap_cvg : 15.85%
unique_crashes : 378
unique_hangs : 0

6 Analysis

6.1 Produced outputs

6.1.1 AFL

In AFL, there are three subdirectories created within the output directory and updated in real time:

- **queue/** - test cases for every distinctive execution path, plus all the starting files given by the user.
- **crashes/** - unique test cases that cause the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT). The entries are grouped by the received signal.
- **hangs/** - unique test cases that cause the tested program to time out.

6.1.2 KLEE

When performing an analysis with KLEE, it generates one output folder containing all files generated by KLEE. These files include:

- Default global files, such as **info** (containing various information about the KLEE run), **warnings.txt** (containing all warnings emitted by KLEE) and **messages.txt** (containing all other messages emitted by KLEE).
- Test files. These are more important to us, since these files contain the test info for each path. The most important ones are the **test<N>.<error-type>.err** files. These files are generated for paths where KLEE found an error and contain information about the error in textual form.

When we take a look at these error files, we can see what input was given to the program (e.g. i=52) and what assertion error the program returned.

6.2 Analysis of produced outputs

As we can see from the results on Problem13 in the previous section, the outputs of AFL and KLEE differ much for each of the three problems. In this case, AFL has found 378 assertion errors, whereas KLEE has only found 28. There is a reason for this; in AFL, crashes and hangs are considered "unique" if the associated execution paths involve any state transitions not seen in previously-recorded faults. If a single bug can be reached in multiple ways, there will be some count inflation early in the process. So it is logical to conclude that AFL has found many errors multiple times, since these errors were reachable from more than one path. One interesting detail (or maybe a funny coincidence) is that for Problem13, the elapsed time for both AFL and KLEE were 8717 seconds. Of course we terminate AFL ourselves, but we actually ran AFL first.

Furthermore, KLEE is a tool that has more coverage than AFL. KLEE is in that sense a lot 'smarter' than AFL, since it looks at the internal program structure in order to find the possible paths, whereas AFL is simply a fuzzer that looks at all execution paths in a mostly random way.

6.3 Future usability of AFL and KLEE

For the future, we foresee that a combination of a fuzzer and a concolic tool will be able to realise the best results. We can see the great effectiveness of a fuzzer like AFL, but to get high code coverage and a reliable result, we think that a tool like KLEE will outperform a fuzzing-only tool. Also because it's easier to include a fuzzer into another testing framework. We have also tested a little bit with angr, which is basically exactly that: a richer framework which realises that the two approaches are complementary, and thus combines concolic execution with fuzzing, resulting in having the best of two worlds in one tool. Due to time constrains, unfortunately we have not been able to perform test runs with angr.

References

[1] M. Zalewski, *American Fuzzy Lop: An open source binary fuzzing tool*, Accessed: 2017-03-21. [Online]. Available: `http://lcamtuf.coredump.cx/afl/`.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

[3] *Rigorous Examination of Reactive Systems*, Accessed: 2017-03-21. [Online]. Available: `http://rers-challenge.org/`.

[4] M. Zalewski, *Readme for AFL*, Accessed: 2017-03-21. [Online]. Available: `http://lcamtuf.coredump.cx/afl/README.txt`.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 1032–1043, ISBN: 978-1-4503-4139-4. DOI: `10.1145/2976749.2978428`. [Online]. Available: `http://doi.acm.org/10.1145/2976749.2978428`.