CS4110 Lab 1

Differences between fuzzing and concolic execution

Authors:
James Sparreboom - 4204751
Julian Faber - 4189736
Marnix de Graaf - 4172949

March 20, 2017

1 Introduction

There are several ways of testing an application. With early testing methods it was required to manually write test cases, often leading to low code coverage and many bugs due to missed test cases. Nowadays test cases can be generated using smart software suites.

One way of generating test cases is to use Fuzzing. With fuzzing random input data is used that is entered into the tested program. For each input the result (crash, hang, succeed) can be turned into a test case.

Another method is concolic execution where every possible path, which can be taken by the program, is executed trying to hit all lines of code. At each branch(e.g. if-statements), constraints (for the inputs) are generated which lead to errors, security vulnerabilities, memory corruption or other kinds of unwanted behavior. These constraints can be transformed into test cases which will have the same effect on the unmodified program.

In this paper both named testing methods will be described in section 2 and 3 respectively. This is followed by section 4 describing the workings and advantages of two tools implementing these techniques: KLEE and AFL. Section 5 shows an analysis on experimenting with the tools on reachability problems. The paper concludes with a discussion in section 6 and a conclusion in section 7.

2 Fuzzing

Fuzzing is a method of testing that creates unexpected or random input data, that is entered into the program under observation.[1] For each input the program can either crash, hang, or succeed. When a crash or hang occurs the used input that caused that result is recorded. By using this data a multitude of errors or vulnerabilities can be found that would most likely not be found by writing manual tests. Fuzzers can be divided into two main categories based on how they create input for the programs, namely mutation and generation-based fuzzers.

Mutation-based fuzzing takes samples of valid input and mutates the input to produce a malformed version of it, which is then entered into the program. One problem is that the input can be malformed in such a way that it is immediately rejected by the program so that it does not give significant results. Smarter fuzzers can solve this problem by making sure that only specific parts of the input are altered or that the overall structure does not change.

Generation-based fuzzers generate completely new input rather than mutating existing input. This is usually done by creating input that has a format that is not immediately rejected by the program, but a completely random input is also considered generation.

An advantage of fuzzers is that it takes little effort to run and can be left running for a very long time to keep finding more bugs. On the other hand it will most likely not find all of the bugs, because some bugs may require very specific input to crash and bugs that do not crash or hang the program will not be caught.

As an example, when running the fuzzing code on the example program below will give an insight in how it would find bugs. Any random input generated by the fuzzer that is not an integer equal to one or two will fail in the switch case and thus cause a crash. When inspecting these crashes the user might discover that the switch case is lacking a default case and is therefore failing.

```
bool isOne(int input){
        switch(input){
        case 1:
                return true;
        case 2:
                return false;
    }
}
```

3 Concolic execution

Symbolic execution in the context of software testing is to explore as many different program paths as possible within a given amount of time, trying to hit all lines of code. For each path a set of concrete input values leading to that path (constraints) are generated and a check is performed for the presence of various kinds of errors such as uncaught exceptions, security vulnerabilities, memory corruption or assertion violations. This allows the creation of high-coverage test suites next to providing developers with the concrete values that triggers a bug.

Concolic execution is a mix of concrete and symbolic execution, where symbolic execution is dynamically performed on concrete input values[2]. During concolic testing a concrete state and a symbolic state is kept: the concrete state maps all variables to their concrete values while the symbolic state only maps variables that have non-concrete values.

Since the concrete state is kept, the execution needs some initial concrete values for its inputs (unlike classical symbolic execution). Starting with some given or random input, concolic testing gathers symbolic constraints on inputs at conditional statements along the execution. It then uses a constraint solver to infer variants of the previous inputs in order to generate another execution path for the next execution. This process is repeated systematically or heuristically until all paths have been tried (or the given time limit expires).

A very simple example where Concolic testing would branch (try another path) and thus create a test case is the following:

```
int divide(int x, int y){
        return x/y;
}
```

A division-by-zero test case is generated for when $y = 0$.

4 AFL and KLEE

This section describes the workings of two tools which implement the techniques discussed in the previous two chapters. How do they work? What are the tools good at?

4.1 What is AFL

American fuzzy lop (AFL) is a mutational fuzzer which uses genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Using these found test cases can improve code coverage substantially for the program under testing.

AFL works by taking a user provided sample command that runs the application and at least one example input file and tries to execute the specified command. If the command succeeds it proceeds to try to reduce the input file to the smallest one that triggers the same behaviour. After that the fuzzer applies various modifications to the input file using the genetic algorithm to get interesting input. Crashes or hangs in any of these inputs may indicate bugs or security vulnerabilities and thus the modified input file is then saved for further user inspection.

What is good about AFL is that it requires essentially no configuration to run and runs relatively fast compared to other fuzzers. Next to that its basic command line user interface gives detailed information on the amount of (unique) crashes and hangs, how longs it has been running, and how much of the program it has covered. The effectiveness of AFL has also been proven by having found a lot of vulnerabilities and non-security improvements since it was first introduced. A few of the systems in which it has found these vulnerabilities or improvements are Firefox, SQLite, and OpenSSL.

4.2 What is KLEE

KLEE [3] is a symbolic execution tool capable of automatically generating tests (inputs), achieving high code coverage on a diverse set of environmentally-intensive and complex programs. The tests it generates can be run on a raw, unmodified version of the checked code (assuming the code is deterministic), after which it will follow the same path and hit the same bug (without getting false positives).

The basic idea is the following: Instead of running code on manually (by the programmer) or randomly-constructed input, run the bytecode on symbolic input which can be anything initially. KLEE has two goals: (1) hit every line of executable code and (2) detect at each dangerous operation (such as dereference or assertion) if there is any input value which could cause an error. To do this it first compiles programs to LLBM assembly language[4], which KLEE can directly interpret and can map instructions to constraints without using approximation. Each symbolic process (or state) has it's own register file, stack, heap, program counter and path conditions. An interpreter loop is executed which selects a state to run and symbolically executes it in the context of that state. If a branch in execution is possible, the state is cloned and execution is performed on both paths with its own states. If an error is detected (or when an execution path reaches an exit call), KLEE solves the current path constraints (or conditions) which trigger the state and turns it into a test case.

What makes KLEE an extra contribution to existing research into symbolic execution test generation is that it tackles the two following common concerns: the exponential number of paths through code and the challenges in handling code that interacts with its surrounding environment (operating systems, storage, the network or the user). Before, they either cannot handle programs that make use of the environment or they require a complete working model. KLEE shows that it can run on a broad range of real code out of the box.

To challenge the exponential number of paths through code, KLEE employs a variety of constraint solving optimizations. The constraint solver normally takes up most of execution time for KLEE so this is an important item to optimize. This is done by expression rewriting (query optimizations), constraint set simplification and using a counter-example cache which eliminates redundant queries. It also represents program states compactly by using copy-on-write at the object level which reduces per-state memory requirements. Using the heap as an immutable map, portions of the heap can also be shared among states, next to being able to be cloned in constant time (which is important given the frequency KLEE does this). As a final optimization, search heuristics are used to get high code coverage. Multiple search strategies for paths to be executed are used in a round-robin fashion, protecting against cases where an individual search strategy gets stuck. All this contributes to a great increase in performance comparing to older methods (as for example KLEE's predecessor EXE [5].

A straightforward approach is used to deal with the surrounding environment. Conceptually it is wanted that all possible values that a read operation could legally produce are returned, rather than a single concrete value. Effects of writes to the environment should be reflected in subsequent reads. This is done by redirecting the calls to models that understand the semantics of the action well enough to produce the constraints. As an example; when an actual concrete file is wanted from disk, the system call is simply invoked to the operation system. For a symbolic file however the operations effect is is emulated on a simple symbolic file system private to each state. The same idea is used for system calls where environmental failures are emulated.

4.3 Experiments on RERS reachability problems

To compare the use of AFL and KLEE, both the tools have been used on RERS reachability problems from the year 2016 [6]. From this set of problems, both tools have been compared on problems 10 to 15 with increasing complexity, in the ‘C99‘ language.

The RERS reachability problems require the internal state machine to be tested, to find out which specified error states can be reached. These error states can be reached by providing different integer inputs created by for example AFL or KLEE. By testing it on the RERS reachability problems, the performance of both the tools can be compared as both will try and reach the error states.

For AFL, the starting input for the tests has been given in 3 files containing ‘”-1 ”‘, ‘”0 ”‘‘ and ‘”1 ”‘‘ respectively. At the start of AFL it is shown that it favors (and therefor uses) only one of the tests files, as all these options are possibly created by the fuzzer itself.

5 Analysis on experiments

The experiments ran on the first five problems of the RERS reachability problems from 2016 showed big differences between the problems itself and the tools. Using both of the tools on the first two problems generated multiple unique crashes and hangs. The last 4 problems did not cause any crashes using the AFL fuzzer after running it for prolonged times (+10 hours). Using KLEE, the execution of the latter problems was done very quickly, finding no or a very few crashes. All runs of KLEE are done until it terminated, except test 11, which has been killed after about 6 minutes, since no new results seemed to be found.

5.1 What do the outputs mean?

AFL generates files as output, including the stats from the fuzzer run, and information about every hang or crash. The main stats file created by AFL contains the general stats of the run, which contain the amount of cycles done, the amount of found (unique) paths through the program and the amount of (unique) crashes and hangs[7]. This data gives a general overview of what has been found during the run. AFL also generates a file for every hang or crash, containing the input given to the program causing the problem.

KLEE generates a binary file (.ktest) for every test it has created, which is readable using the program ‘ktest-tool‘. By using this tool on one of the files, information is given on the program tested, such as its name and argument, but the most important is the ‘data‘, which contains the input given to the program in that test. Besides the binary files for every test, KLEE also generates human-readable files containing general statistics of the run, including timers and the amount of covered branches in the program. This data can be retrieved easily by using the ‘klee-stats’ tool, which then shows the stats ‘ICov’ and ‘BCov’. These statistics are respectively; the percentage of the instructions covered and the branches covered. The amount of branches covered can be compared to the AFL statistic for map coverage, which also shows the percentages of branches covered.

Both AFL and KLEE create tests/input, which can be used to determine what causes the bug. This allows the programmer to search for the problem in a structured way. This can for example be done with the KLEE replay library or by simply running the program with the input received from the results. The KLEE replay library allows the developer to replay a test case in a non-symbolic manner, which then can be used to find the location of faulty instruction.

5.2 Outputs

The AFL statistics are as shown in table 1.

Table 1: AFL statistics

Problem#	Executions	Time (sec)	Found Paths	Bitmap Cov	Crashes	Hangs
Problem10	23507007	3899	123	1.37%	67	2
Problem11	18372448	5760	656	2.33%	106	2
Problem12	29699434	5284	91	0.76%	0	1
Problem13	263496952	38022	12	0.36%	0	2
Problem14	254319602	36452	13	0.55%	0	1
Problem15	58261407	9210	35	0.74%	0	1

It can be seen that not all test have ran during the same time. Problem 10 and 11 have been ran until the time between found crashes became less than 5 minutes between each other. The other problems have been ran for a longer time, especially problem 13 and 14 which have been running for about 10 hours. These problems did not find any crashes, but also have a low bitmap coverage. This means that the fuzzer did not find many paths through the program, or that the program does not contain many paths.

The KLEE statistics are shown in table 2. The results from the running KLEE on the problems

Table 2: KLEE statistics

Problem#	Instruction	Time	ICov	BCov	ICount	TSolver
Problem10	69994485	495.35	90.10	71.55	4877	8.43
Problem11	70057266	382.26	88.6	66.36	15182	7.6
Problem12	72741	1.19	3.86	4.41	75219	90.64
Problem13	1896	0.16	0.84	2.64	224925	41.39
Problem14	2322	0.16	1.12	3.05	206971	28.56
Problem15	7793	0.45	0.53	1.06	562422	70.91

show that the first two problems have a high coverage. All other problems terminated execution within a short time, also having a low coverage. This could be caused by the fact that many paths were unreachable.

In the first two problems, the instruction coverage (ICov) is high, as is the branch coverage (BCov). From this can be deduced that nearly all instructions and branches have been reached by KLEE. In the other problems these values are much lower, which also explains the short execution time.

6 Discussion

The outputs from both AFL and KLEE were significantly different between the first 2 problems and the other programs. While this could be a result from having more unreachable code or a structural difference between the problems, it could also be caused by a mistake in making the original code work with AFL and KLEE, or a mistake in parameters. For AFL a memory limit of 250MB and time limit of 2000ms have been used. KLEE has been run with a memory limit of 2000MB and no time or depth limit.

Since KLEE can be used on any real unmodified program its usability is much greater than older methods which needed adaptations to the code or was not able to fully test the program. Testing can be run quick and it reveals lots more errors than traditional testing methods. This makes it an ideal tool to use to find bugs the programmer did not think of and prevent security bugs.

7 Conclusion

Both fuzzers and concolic execution can be used for advanced testing of programs, by generating test inputs for programs in a way humans would be unable to do in reasonable time. Where a fuzzer (for example AFL) generates tests by mutating a given example input or by using random input, concolic execution (for example KLEE) creates tests using concrete and symbolic states and variables.

While both can be used for the same purpose, there is a difference in the output and uses of the tools. AFL can be used for extended periods, to allow it to find all possible crashes and hangs in the program, but it works on all types of programs. KLEE on the other hand terminates when it has finished executing all possible paths which could be found, but it requires the code to be in LLVM bytecode.

References

[1] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: http://doi.acm.org/10.1145/96267.96279.

[2] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[4] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[5] Cristian Cadar et al. "EXE: automatically generating inputs of death". In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), p. 10.

[6] B Steffen. "http://rers-challenge.org/2016/". 2016 (accessed March 12, 2017).

[7] *AFL Status Information*. "http://lcamtuf.coredump.cx/afl/status_screen.txt".