Reverse Engineering – Assignment 1

Software Testing and Reverse Engineering

E. Geretto (1869426)
A. Toptsoglou (1823450)
G. Iadarola (1879480)

March 20, 2017

1 Introduction

This report gives, at first, a general introduction about fuzzing and concolic execution, in Section 2 and Section 3, and then describes the results we obtained applying those techniques to the RERS challenges, in Section 6. Section 7 describes instead the attempt to use the `angr` framework to solve the same problems.

2 Fuzzing

Fuzzing is a software testing technique whose purpose is to find bugs in programs; the tools which perform fuzzing are the so-called fuzzers. Fuzzers try to provide different inputs to the software that is being tested and report which of those generates a crash. In a sense, fuzzers are brute-force vulnerability scanners. These programs should be let running for a large amount of time in order to give valuable reports. The whole process is automated and the user should only provide a test case which represents a valid program input.

The efficiency of a fuzzer is based on how the inputs are generated. Although the simplest implementation of this technique is random fuzzing, where the input is chosen randomly, the efficiency of the process is low because the chances of finding a suitable input are limited. [5] One very efficient type of fuzzing is the genetic one in which the selection of the inputs is guided by a genetic algorithm.

In genetic fuzzing, the generation of new inputs is guided by a fitness function which, as a consequence, determines the quality of the algorithm itself; the most common approach is to use a measure of the likelihood of finding new paths for a given test case.

In general, a genetic algorithm consists of several iterations; in each one of them, the current set of individuals, called current generation, is used to obtain the following one. Applying this definition to fuzzing, the test cases that are currently used are mutated in order to obtain new ones. This mutation is obtained through implementation dependent operations that modify the elements that constitute an individual, called chromosomes. In the case of fuzzing, in which the bits of a test case are its chromosomes, common operations are bit flips, insertions, deletions and cross-overs.

Once the new individuals are generated, their fitness is graded using the aforementioned fitness function and compared to the one obtained by the individuals belonging to the current generation. After this, the new generation is obtained selecting the individuals with the highest fitness; those left out are simply eliminated. In this way, the new generation will be always at least as good as the previous one.

However, genetic algorithms can suffer from solutions that seem to be good enough but are not the best. This is the known problem of finding a local maximum instead of the global maximum. Genetic algorithms try to avoid this problem by inserting random individuals, far from the current optimal solution, to every new generation hoping that those can hit near the global maximum and consequently move the search away from a local one.

The following example illustrates how a genetic algorithm works. Let f be a fitness function equal to the opposite of the distance from 5, so that its

global maximum is indeed 5. This can be represented as:

$$f(x) = -|x - 5|$$

In this case the individuals are numbers composed by bits which are, in turn chromosomes. The first step is generating five random numbers. The fitness function will grade these five numbers and the mutation process will then generate five more. At this point, the ten numbers will be graded again. The best five will form the new generation while the others will be deleted. After this, the process is repeated. Once number 5 is present in a generation, the search will be blocked on the global maximum.

Of course genetic fuzzing, as all fuzzing techniques, is not suitable for testing all kind of programs. Programs, such as parsers, that receive multiple inputs are suitable for fuzzing because many different paths of the code can be explored easily. However, in case of programs that accept a complex or a restricted input the fuzzer will not be efficient. Indeed, it will keep trying to find the correct "restricted" input in order to be able to explore new paths but it will be stuck due to the large search space available failing to explore the whole program. [5] For this reason, in this kind of problems a good alternative is concolic execution which will be described in Section 3

3 Concolic execution

The second technique that has been used to produce the results presented in Section 6, is called *concolic* execution. This term derives from the combination of two words, *symbolic* and *concrete*, which reflects how the technique is structured.

The starting point is indeed *symbolic* execution, which involves running the program using symbolic variables, not concrete ones, that are expressed using arithmetical formulas and that can be constrained with conditions. Every time a new variable is introduced in the program, its value is expressed either as the combination of those already present or it is left unconstrained. Moreover, when it is modified, the expression defining the variable is modified accordingly.

In addition, when a branch condition is encountered, an SMT solver is interrogated in order to understand if the branch condition can be satisfied given the current constraints on the symbolic variables used in it. If the result of the SMT solver is positive, the condition is added as a constraint for that path and the branch is taken. Moreover, if also the opposite of the examined condition can be satisfied, then that path should also be taken; in order to do so, a new execution thread is spawn to follow the path where the condition is met, while the current one is used to follow the path in which the condition is not met. Obviously, in both cases, the branch condition is added to the general path condition that summarizes all the branches taken to reach that particular point in the program.

When the target of the research is reached, for example a crash is met or a vulnerable statement is executed with an unconstrained value as argument, the symbolic execution engine usually outputs a test case that can be used on the original program in order to reach the marked statement. This test case is obtained using the SMT solver to resolve the current path condition producing concrete values for all the symbolic variables present. [2]

```
1 int get_sign(int x) {
2   if (x == 0)
3     return 0;
4
5   if (x < 0)
6     return -1;
7   else
8     return 1;
9 }
```

Listing 1: A function that returns the sign of the argument.

As an example of this process, Listing 1 can be considered. If variable `x` is passed, unconstrained, to the function, the first branch condition at line 2 will spawn a new process given that `x` can be either equal to 0 or not. In this case, the execution thread inside the first branch will terminate immediately generating a test case respects the path condition `x == 0`, which has to be met to reach line 3. The other execution thread will instead reach line 5, where a new branch condition will cause another split; indeed, both the positive and the negated condition for this branch can be met. As a consequence, a new thread will be spawn to follow the positive condition and the current one will be used to follow the negated one. In both cases, the return statement will be reached and the threads will terminate generating two test cases: the first

one will satisfy the path condition `x != 0 && x < 0`, so that line 6 can be reached, and the second one will satisfy `x != 0 && x > 0`, which allows to reach line 8. In order to obtain the corresponding concrete values for `x`, an SMT solver can be used. [7]

As far as the *concrete* part of the execution is concerned, it was introduced in order to make symbolic execution feasible on real life programs. Indeed, programs are not usually limited to linear conditions that can be solved quite easily with a SMT solver, they may interact with non-symbolic code, call system routines or use complex mathematical operations that cannot be easily reproduced.

In order to solve this problem, when such a situation is encountered, an SMT solver is used to concretize the variables used in the considered expression according to the path condition. Moreover, the concrete result of the expression itself is used for the rest of the execution. Concrete and symbolic variables are mixed in the following way: when only concrete values are considered, then the operation is done concretely, but even if just one symbolic value is used, then the value is calculated symbolically. This solution allows symbolic execution to be applied on normal programs in a reasonable amount of time, but sacrifices its intrinsic completeness. [2]

As compared to fuzzing, presented in Section 2, concolic execution is more efficient in exploring code that has a series of single, precise, conditions that are difficult to meet with the random mutations that a fuzzer uses. This ability, though, comes at a cost; indeed, when code that accepts a wide range of possible different inputs is examined, for example in the case of a parser, a concolic execution engine is forced to spawn a large amount of threads in order to follow all the possible paths rapidly exhausting all the available resources. In this case, obviously, a fuzzer results way more efficient. Another closely related problem arises when loops are encountered: in this case, a concolic execution engine is forced to spawn a new thread for every subsequent iteration in the loop. There are methods to mitigate this problem, as limiting the number of iterations manually, but they limit the amount of code that will be covered. [5]

4 AFL and KLEE

4.1 AFL

AFL stands for American fuzzy lop and it is a genetic fuzzer. To start the fuzzing process one should compile the code with a modified version of `gcc` designed especially for AFL, the `afl-gcc`. This modified version will provide instrumentation for two purposes. First AFL will be able to count the branch transitions to evaluate the coverage and second the detection of software crashes will be possible. After compiling with `afl-gcc`, the user should provide a test case and run AFL. During the fuzzing, AFL displays a GUI which provides useful information such as how many crashes have been reported, how much time has passed since the last new path exploration etc. When the fuzzing is terminated all the crash reports are saved.

AFL is known for its ability to find and report crashes. This is possible because AFL utilizes a large variety of features. The most important one is the genetic fuzzing algorithm which is responsible for the new generation of inputs. The genetic algorithm that is used contains a fitness function, which has the ability to grade the new generation of inputs based on the new paths that these inputs can explore. Thus, every next selected input will, at least, cover a path that previous inputs did not. With the state transition tracking, the genetic algorithm will prioritize the next generation of inputs which are likely to make the program follow a different execution path. AFL also handles the large amount of paths that can be created due to loops. Specifically, with the loop bucketization feature and the usage of some heuristics, AFL reduces the times of exploration, during the iterations, from N to $log(N)$ paths. [5]

Although AFL uses great features for crash reporting, as all the fuzzing techniques it can find difficulties discovering bugs. The main problem is that, when a program requires a specific and complex input, it is very difficult to find. This is where concolic execution programs, as KLEE, can help.

4.2 KLEE

KLEE is a symbolic execution tool which follows the procedure described in Section 3 in order to produce high coverage test suites for the program

considered. In particular, it works as an interpreter for LLVM bytecode, which helps to simplify the execution process while still being close to assembly code. For this reason, the programs examined, while not requiring any modification, need to be compiled using this intermediate language.

One of the main improvements that KLEE introduced over plain symbolic execution is the thorough emulation of the environment in which the program runs. Indeed, the authors of the tool were able to create a simple symbolic file system that allows to pass files as symbolic data and to impose constraints on them. This file system is also well integrated with the normal one allowing KLEE to provide normal files when the execution requires it. In order to provide this functionality without completely rewriting the C standard library, KLEE uses an LLVM compiled version of the uClibc library and emulates only the system calls made to the operating system deciding dynamically if they should be emulated or passed to the kernel.

Another important feature that KLEE provides is the reproducibility of the crashes through the tests generated. This is made easy by a library provided by KLEE which can be compiled together with the original code; indeed, it allows to reproduce the execution environment and the inputs using the binary test cases produced by the tool, eliminating the need for doing it manually.

After the completion of the symbolic execution, it is quite common that the execution of KLEE over a certain codebase produces a test suite that reaches a coverage which is higher than the one provided by the suite written by the developers themselves. This result was achieved, for example, on the `coreutils` set of executables. [1]

5 Experiments

This section is about the experiment set up to show the differences between fuzzing and concolic execution. Two tools were used to perform the tests, the AFL [8] for fuzzing and the KLEE [6] for concolic execution.

The AFL and KLEE were run on a set of reachability problems, taken from the RERS challenge edition 2016 and 2017. [3] Precisely, the problems were the 10th to 18th for the 2016 edition and the 4th to 6th for the 2017 edition.

The C code provided by the reachability problems contained a high number of branches and possible paths that lead to different points in the code, marked as *errors*. Only a small set of them is reachable and the challenge consists in finding the input that leads the software to these points.

The source code of the problems was edited in order to make it runnable for AFL and KLEE. These changes were required to link the input and output of the two programs, the tools and the compiled problems.

As a fuzzing tool, AFL randomly looks for new inputs and branches, so it could run forever. Thus, we stopped it manually but at least one cycle was completed on all the problems. As far as KLEE is concerned, it proved to be faster than AFL. In this experiment, KLEE stopped automatically on almost all the problems, except for the 17th. In this unique case, it was manually stopped after 2 hours and half, when the instruction coverage percentage remained stable for more than 20 minutes.

As a group project, our first concern was to organize the project environment properly. All three members of this group worked on GNU/Linux environment, which made easier to share code and collect data across our machines. First of all, several bash scripts were written to set up the same environment in all our machines. Then, more scripts were used to collect data and share the results. The experiment was performed on three different machines but exactly the same steps and configurations were used on every host, both for installing and running the tools.

The KLEE tool was run in a docker container while the AFL tool was installed on our machines. It is worth mentioning that we edited the `__VERIFIER_error` function for KLEE as an `inline` function, in order to improve the efficiency and readability of the output for our purpose.

```
1 __attribute__((always_inline))
2 inline void __VERIFIER_error(int i) {
3     fprintf(stderr, "error %d", i);
4     assert(0);
5 }
```

Listing 2: Inline function introduced in C code for KLEE computation.

In so doing (Listing 2), every time this function is called, KLEE produces a test case marked as crash. Without this modification, only the first time the

function is encountered the crash is marked, after that it is simply ignored and thus the test cases need to be checked one by one.

Pointing out more changes, we also added the `--only-output-states-covering-new` option to the KLEE command line. By doing so, KLEE outputs only test cases for paths which either covered new instructions in the code or hit an error. By default, KLEE would write out test cases for every path it explores. Once the program runs for a while, these outputs become less useful because many test cases will end up exercise the same path, and computing each one wastes time. For instance, the final line of the output showed that even though KLEE explored almost ten thousands paths through the code, it only needed to write 57 test cases.

6 Analysis

The collected data were analyzed and this section contains the result of the experiments. Table 1 shows the errors found by the tools. The second and third columns show the unique errors found by AFL and KLEE respectively, while the fourth one the errors discovered by both tools.

Problem	AFL only	KLEE only	Both
10	0	35	10
11	0	7	9
12	0	9	11
13	0	5	21
14	0	10	17
15	0	13	15
16	0	7	22
17	19	0	3
18	0	8	22
4	0	9	11
5	0	17	7
6	24	0	1
Total	43	120	149

Table 1: Errors found by AFL, KLEE and both.

The table clearly shows that KLEE worked better on the experiments. Except on two cases (Problem 17 and 6), KLEE was able to find more errors than AFL. Indeed, AFL found just a small set of errors, which are included completely in the KLEE set. Figure 1 shows this result, every problem has the same weight in determining the final percentage, so that larger problems do not have an excessive influence.

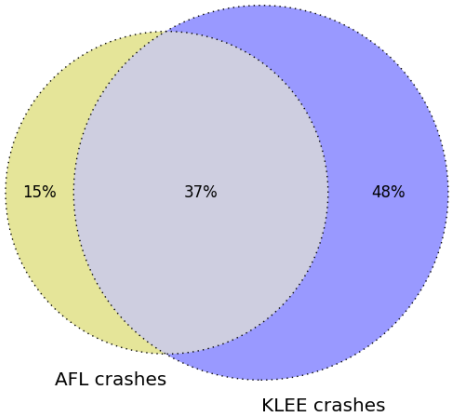

Figure 1: Weighted Venn diagram of the crashes found.

Predictably, KLEE worked better. The concolic execution was able to resolve the branches contained in the code better than fuzzing. The access branches were evaluated by the KLEE solver, which generated specific inputs in order to produce a higher instruction coverage. On the other hand, fuzzing tried to improve the branch coverage using random generated input. Theoretically, AFL may have been able to cover the same instructions as KLEE, but the process could have lasted forever. As long as the conditional statements number and the variables are manageable by the constraint solver, concolic execution will be more efficient than fuzzing.

The difference between the two tools was caused by the limitation of the concolic execution, as briefly described in Section 3. While AFL, randomly looking for errors, has found a constant number of errors in each problem, KLEE has had issues in some of them because of the path explosion. Indeed, KLEE was not able to find many errors on Problem 17 and Problem 6, which are way bigger than the others. For instance, Problem 17 contains 36 MB of code while the average of the other problem is around 5 MB, same for the array of the input that contains way more variables than the other problems. Therefore, this code includes many branches and conditional statement which lead the

KLEE solver to face path explosion in order to test all the possibilities.

On the other hand, AFL was not affected by the higher complexity of the problem. Fuzzing does not face problems like the path explosion because it does not force the computation to cover all the paths. AFL aims to cover as many paths as possible with random mutations in the inputs. Due to this behavior AFL is not as efficient as KLEE but, on the other hand, it does not exhaust all the available resources. Hence, it is suitable for problems where the computational power of the machine is not enough to handle KLEE or other concolic execution tools properly.

However, the computational power will probably increase exponentially in the next years and concolic execution could move forward in managing path explosion issues. Nevertheless, the solution proposed by Driller [5], which leverages both fuzzing and concolic execution, will keep being a great answer. Driller uses selective concolic execution to explore only the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer either is unable to satisfy or it would satisfy after a long time. By combining the strengths of the two techniques, it mitigates their weaknesses, avoiding the path explosion inherent in concolic analysis and the incompleteness of fuzzing.

7 The `angr` framework

7.1 General description

The `angr` framework is a set of Python modules that can be used to build interesting binary analysis scripts. One of the main features is that it allows to analyze compiled binaries; indeed, it does not require the user to recompile the code in LLVM bytecode, as KLEE does (Section 4.2), nor it needs instrumentation to improve the performance of the analysis, as AFL (Section 4.1). [4]

The ability of analyzing binaries is really useful in order to reverse binary blobs, as for example the firmware of an IoT device, or in other situations in which the code is not provided. However, it is worth mentioning that, for this particular case, working on compiled binaries is not the best possible choice due to the fact that all the additional, and helpful, information provided in the source code is lost.

The framework implements several types of analyses, both *static* and *dynamic*: as far as the former are concerned, the most relevant are the recovery of the *Control Flow Graph*, which allows to have an overview of how the binary is structured, and the *Value-Set Analysis*, which instead tracks all the possible values that a variable can assume at a particular point in the program. [4]

As far as dynamic analyses are concerned, the most important is dynamic symbolic execution which, in `angr`, can be executed not only from the entry point of the program but also from any point in the code, for example just inside a single function; in this second case, it is defined as under-constrained symbolic execution. The execution engine has been implemented relying on the VEX intermediate representation, originally created for `valgrind`, and uses the Z3 SMT solver to resolve constraints inside the conditional branches.

The authors of the tool used this framework to write several other tools, one of which is called Driller. This particular tool leverages the symbolic execution engine in `angr` to guide the fuzzing process operated by AFL.

7.2 Reachability problems

Considering that the goal of the analysis on the RERS problems is to evaluate the reachability of a particular piece of code, the only suitable analysis between those previously presented is dynamic symbolic execution.

The symbolic execution algorithm implemented in `angr` is quite similar, as explained by the authors, to the one used by KLEE. The only difference is that, being written in Python instead of C++, the speed is obviously reduced enormously. This design choice was made probably because that particular engine was designed to be used for symbolically execute small chunks of code, instead of the whole program. Moreover, the goal of the authors was probably to evaluate the presence of a path to reach a certain location, not to obtain all the paths that can reach it, which results in a computation that is way more expensive in terms of resources.

Nonetheless, it was possible to realize a small script that extracts the address location of the

__VERIFIER_error function, target of the analysis, and leverages the PathGroup interface, which allows to manage bulk symbolic executions, to explore the binary. After that, for each found path, the stderr stream is extracted in order to print the output of the function considered. The symbolic execution instance can then be relaunched in order to obtain the next path.

The execution of such script, however, failed to produce the expected results. As stated previously, angr relies on Z3 to solve the constraints encountered in the code. However, it uses a custom compiled version of this SMT solver and, for that reason, its reliability was compromised: after producing just a few outputs, a segmentation fault was generated inside libz3.so and, unfortunately, the problem could not be solved.

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] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[3] The RERS committee. The RERS challenge. http://rers-challenge.org/. Accessed: 2017-03-13.

[4] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[5] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[6] The KLEE team. KLEE—LLVM execution engine. http://klee.github.io/. Accessed: 2017-03-13.

[7] The KLEE team. KLEE—Testing a small function. https://klee.github.io/tutorials/testing-function/. Accessed: 2017-03-11.

[8] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2017-03-13.