STRE - Assignment 1

Ricky Sewsingh - 4230981
Julian Hols - 4247930

March 2017

1 Introduction

This report is evaluating the efficiency of the tools *AFL* and *KLEE*, by experimenting the tools on the RERS 2017 problems [1]. AFL is a *fuzzing* tool and KLEE *concolic execution* tool. First a small explanation is given on fuzzing and concolic execution. Then a description is given off AFL and KLEE. Then we explain how we used the tools on *RERS* problems, explain the results and the usability of the tools. Lastly we compare the results of the two tools and try to give an explanation of the differences.

2 Fuzzing

Fuzzing is an automated software testing technique for computer programs. It works by feeding a program with input that is automatically generated by the fuzzer. The inup can be invalid and unexpected or even completely random. A fuzzer will monitor the program's behaviour and results. It and has the goal to find runtime errors, crashes and hangs using the generated input. Fuzzing is often used to test security-critical software for possible vulnerabilities that might be exploited. It can also be used to find bugs in software systems but finding no bugs does not prove their absence.

Take the following example.

```
int main(unsigned int argc, char** argv) {
        char buffer[256];
        bool allowed = false;

        if(argc > 3){
                allowed = strcmp("password", argv[2]) == 0;
                sprintf("The password you specified was: '%s' and is %s!",
                                argv[2], allowed ? "correct" : "incorrect");
                printf(buffer);
        }

        if(allowed){
                printf("The secret is ...");
        }
}
```

A fuzzer can generate random input for this program and can by chance find the password but more likely will find that the program crashes if it feeds it a large input. This is because the program contains a buffer overflow if the argv[2] input is larger than 256 bytes. The fuzzer will save the generated input that has caused a crash. The user can now reproduce the crash and perform further analysis on the it and find that the program has a vulnerable bug.

There exist fuzzers of different levels of sophistication and may have different techniques of input generation. Random input generation can be used, this can for example be used for black box testing where no information about a program's input is known. A fuzzer can make use of an existing input seed. For example by using an existing input file and applying mutations to it. A fuzzer can be aware if the input structure, to generate inputs that are valid program inputs. Or a fuzzer can analyze the program in order to generate input and tries to reach maximum code coverage. An example of a fuzzer that uses this technique is AFL, which uses compile time instrumentation to keep track of available paths and paths that it has reached.

3 Concolic execution

Concolic execution combines the aspects of both symbolic execution and concrete (normal) execution, hence the name. For symbolic execution, program variables are treated as symbolic variables. For concrete execution, the program is tested in particular inputs. For concolic execution, the concrete execution drives the symbolic execution. The program is given some input, but it also maintains symbolic information. This way, during execution, when it reaches a branch, it can decide to enter the another branch, by asking an SMT solver to provide an input that enters the branch. For example, take the following code example.

```
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);
}
```

First random input is used, suppose x = 14 and y = 5. Now the execution will take the 'else' branch of the expression 'if (x == z)'. The concolic execution system might want to trigger the 'true' branch. It does this by calling an SMT solver to provide a set of inputs to trigger the 'true' branch. Suppose, x = 2 and y = 1. Now it goes into the 'else' branch of the expression 'if (x > y + 10). Again the concolic execution might want to trigger the 'true' branch. Again the SMT solver is asked to provide a set of inputs to trigger this branch. Suppose x = 40 and y = 15. Now the program reaches the ERROR state. This example shows how the concolic execution method keeps symbolic information to determine new inputs for the concrete execution to use.

4 AFL and KLEE

4.1 AFL

AFL stands for American Fuzzy Lob. It is a toolset that can perform security-oriented fuzzing for programs written in C, C++, or Objective C, compiled with either gcc or clang. The tool afl-fuzz works as an intelligent file-format fuzzer that makes use of compile-time instrumentation and genetic algorithms to automatically discover paths (test cases) in the program that it did not cover yet. It optimizes each cycle to take advantage of all the new code paths it has found. AFL begins the actual process of fuzzing by applying various modifications to the input file and running the program with the generated input. If a program crashes or hangs, AFL will store the used input file so that the user can perform further inspection, as this might suggest the discovery of a new bug or possibly a security vulnerability.

Programs are compiled using the provided afl-gcc or afl-clang to perform the compile time instrumentation. The generic fuzzer is capable of synthesizing complex file semantics in a wide range of targets. As a result of its generic use there is no need for special purpose-built and syntax-aware tools. AFL has the advantage over other fuzzers that it is fast, as it has a very low performance overhead. The use of low-level compile-time instrumentation as well as various optimizations results in near native fuzzing speeds against real-world targets. The tool is designed to be simple to use and requires almost no configuration. Disadvantage of AFL is that it only runs on macOS and Linux and only supports C, C++, and Objective C. However there exits forks and derivatives of AFL that allow support for other languages and use on windows systems.

4.2 KLEE

KLEE is a concolic execution tool, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs [2]. KLEE tries to hit every line of executable code in the program and detect at each dangerous operation if any input value exists that could cause an error. KLEE does this by running programs symbolically. Operations generate constraints that exactly describe the set of values possible on a given path. If KLEE detects an error or if a path reaches an exit call, KLEE solves the current path's constraints, its path condition. To solve a constraint, KLEE uses its own constraint solver. The result is a test case that follows the same path when the program is rerun. This way KLEE tries to execute all possible paths that the program can take.

5 RERS reachability problems

We have decided to perform experiments of both AFL an KLEE on the RERS 2017 reachability training problems. The 2017 challenge contains three reachability problems: Problem4, Problem5 and Problem6. All problems contain the problem program it's source code that has been highly obfuscated to make analysis harder. The goal of the challenge is to figure out which VERIFIER_error(s) are reachable in the program.

5.1 AFL

First we need to make some slight changes to the source code for each of the problems in order for it to work with the input generated by AFL. We have added a local implementation for the __VERIFIER_error(int); method and changed the main function to use the input generated by afl-fuzz.

We can now compile the program using either afl-gcc or afl-clang.

```
afl-gcc Problem4.c
```

These will compile the source code and perform the compile time instrumentation that allows afl-fuzz to track control flow of the program.

The last thing we have to do is to create a sample input file.

```
mkdir input
printf "\"1\n\"\n" > input/input.txt
```

Now everything is in place to start fuzzing, for each of the problems we start afl-fuzz.

```
afl-fuzz -i input/ -o output/ ./a.out
```

5.1.1 Results

We let the fuzzer run for 3 to 4 hours for each of the problems. In table 1 you will find the overall results from afl-fuzz.

Problem	Fuzzing time	Cycles done	Total paths	Unique crashes	Unique hangs
Problem 4	3 hrs, 42 min	6	926	46	19
Problem 5	3 hrs, 37 min	13	551	98	6
Problem 6	5 hrs, 59 min	163	741	142	22

Table 1: Overal results

The output directory for each of the problems contains the following content:

- crashes, this directory contains all the files that were used as input (unique test cases) that caused the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT).
- hangs, this directory contains all the files that were used as input (unique test cases) that caused the tested program to time out.

- queue this directory contains all the files that were used as input (unique test cases) that result in a distinctive execution path.
- fuzz_bitmap, a file to keep track of the number of times a particular branch within the fuzzed program has been taken.
- fuzzer_stats, contains statistics about the run such the information show in table 1
- plot_data, file containing data used by afl-plot to generate a plot about the fuzzing progress.

We are interested in all the inputs that caused a crash as these are a result from input reaching an error function. To find the error function that is reached with a certain input we run the program for each of the input files in the crashes directory. We wrote a simple bash script that takes a problem directory as its argument and prints all the errors that the program produces using the inputs in the crashes directory.

```
#!/bin/bash
for filename in \$1/output/crashes/id*; do
    cat \$filename | \$1/a.out | grep error;
done
```

If we store all the output in a file called results.txt we can list all the reachable errors in a nice way using:

```
cat results.txt | sort | uniq
```

You can find the reachable errors that we found using AFL fuzzing in table 2.

Problem	Found reachable errors	Missing reachable errors
Problem 4	3 12 25 33 35 38 47 50 51 54 61 68 71 74 77 80 81 88 90 93	none
Problem 5	0 4 5 6 7 13 14 18 31 35 36 40 41 42 46 52 54 61 62 66 68 69 76 80 86 91 92 94 97 98	11
Problem 6	0 2 8 9 11 15 16 18 19 22 26 28 30 31 32 33 36 40 44 46 57 60 62 64 65 66 67 74 83 84 97 98 99	42 56 73

Table 2: RERS reachable errors using AFL fuzzing

5.1.2 Usability of AFL

We are impressed by the results of our experiment. Using AFL we managed to only miss 4 errors while running the fuzzer for only 6 hours. The generic usability, simple usage and good performance of AFL makes it an attractive tool to use for future projects and on real world programs.

5.2 KLEE

Just as with AFL, KLEE also needs some slight changes in the source code. Again a local implementation for the _VERIFIER_error(int) is added. In order to find all unique errors that has been triggered, we added a piece of code that if an error is triggered and it has not been triggered before, it writes it to a file. This not trivial to do with the output provided by KLEE. Also an assert is added, which causes a crash. This is what we want to find using KLEE. Lastly the main function is changed so it utilises KLEE.

In order to run KLEE, a docker container needs to be created containing KLEE. Then the code is compiled using clang and the resulting code is run using KLEE. We ran KLEE until it is done, but the first Problem tried ran for quite some time, so this was stopped after around 5 hours.

5.2.1 Output

The output directory can be divided in three parts: Standard Global Files, Other Global Files and Per-path Files. A description of the output per part is given below.

Standard Global Files

- **Info**: This file contains various information related to the KLEE run. This includes the command used to run KLEE and the total time taken by the execution.

- **warnings.txt**: This file contains all warnings emitted by KLEE.
- **messages.txt**: This file contains all other messages emitted by KLEE.
- **assembly.ll**: This file contains a human readable version of the LLVM bitcode executed by KLEE.
- **run.stats**: This file contains various statistics emitted by KLEE.
- **run.istats**: This is a binary file containing global statistics by KLEE for each line of code in the program.

Other Global Files

- **all-queries.kquery**: This file contains all the queries KLEE made during execution in KQuery (KLEE's query format) format.
- **all-queries.smt2**: This file contains all the queries KLEE made during execution in the *SMT-LIBv2*.
- **solver-queries.kquery**: This file contains all the queries passed to KLEE's underlying solver during execution in the KQuery format.
- **solver-queries.smt2**: This file contains all the queries passed to KLEE's underlying solver during execution in the SMT-LIBv2 format.

Per-path files (Note: N stands for a number)

- **test[N].ktest**: This file contains the test case generated by KLEE on a certain path.
- **test[N].[error-type].err**: This file contains information about an error in textual form. This is generated if KLEE finds an error.
- **test[N].kquery**: This file contains the constraints associated with a given path, in KQuery format.
- **test[N].cvc**: This file contains the constraints associated with a given path in CVC format.
- **test[N].smt2**: This file contains the constraints associated with the given path in SMT-LIBv2 format.

The interesting things we found in the KLEE output are the .ktest and .err files. If an error has been triggered, a .err file is created. This shows which line got triggered and the stacktrace. The corresponding .ktest file can then be used to replay the inputs. This way you can see what happens when the corresponding input is used and how the error is triggered

5.2.2 Results

Using the following command, we can obtain coverage information of the run.

```
klee-stats ./klee-out-0/
```

The results of this information for each problem can be found in table 3.

Problem	Time	# of Instructions	Instruction Cov	Branch Cov	# of Static Instructions
Problem 4	5 hrs, 33 min	662503172	91.17 %	79.66 %	6028
Problem 5	0 hrs, 37 min	220746931	41.45 %	37.15 %	54814
Problem 6	0 hrs, 5 min	24305369	33.21 %	22.87 %	65965

Table 3: Overal results

Now by using the file containing the unique errors triggered (See beginning of this subsection), we can derive the information of table 4.

Problem	Found reachable errors	Missing reachable errors
Problem 4	3 12 25 33 35 38 47 50 51 54 61 68 71 74 77 80 81 88 90 93	none
Problem 5	0 4 5 6 7 13 14 18 31 35 40 41 42 46 52 61 62 68 69 76 80 86 97 98	11, 36, 54, 66, 91, 92, 94
Problem 6	0 8 11 16 18 19 22 26 28 30 31 32 33 36 40 44 46 60 62 64 65 66 67 74 83 84 97 98 99	2 9 15 42 56 57 73

Table 4: RERS reachable errors using KLEE

5.2.3 Usability of KLEE

KLEE works a lot faster than AFL, but did miss more errors. KLEE is trivial to use, just create a docker container, compile the code with clang and run KLEE. But obtaining useful information from the generated output is a bit harder. There are a lot of files generated, since for every testcase atleast one ktest file is generated. In our example we tried to find all errors that were triggered by KLEE. The messages.txt file only prints the first time it triggers the error and then ignores it. So we have to search for .err files. But these files do not contain the number of the error generated, just a stacktrace. So we have to run the corresponding .ktest file to determine the error generated. Doing this for around 30 errors is quite timeconsuming, considering this can be done way faster. That's why we made our own solution to this issue. Next is the fact that the *info* file gave close to information, even though the documentation says it should have. The next issue was regarding problem 4. Because this was the first problem we tried, we let it run until it stops, but after 5 and a half hours, we believed that it would not stop. However since the other problems did stop, within a short time compared to 5 hours, we believe it could have been a configuration error, but we are not certain that this is the case. The last issue we encountered was with the last problem. It ran for only 5 minutes, which seems way too short. But we could not find the reason why this was the case. It also did not find 7 errors, where AFL did not find 3 errors.

We believe KLEE has much potential if used correctly. The output is very unclear, since there is so much, but this also makes it possible to retrace everything KLEE has done.

6 Comparison

The detection rate of AFL is higher than for KLEE. AFL was able to find more errors compared to KLEE as can be seen in table 5. However the execution time of AFL is way higher than that of KLEE which can also be seen in table 5. We believe this occurs because KLEE does directed testing. It searches for paths to take and tries to find a path to the error. AFL also does some sort of directed testing, but not as much KLEE. AFL takes more of a brute-force approach. So the longer AFL is ran, the higher the chance is to get a higher error detection rate. But since KLEE is more directed in its testing, it has a much lower execution time than AFL, but still manages to trigger a reasonable amount of errors. KLEE will stop after a certain amount of time and give you a concrete result of the errors detected.

Problem	AFL runtime	KLEE runtime	AFL found errors	KLEE found errors
Problem 4	3 hrs, 42 min	5 hrs, 33 min	20	20
Problem 5	3 hrs, 37 min	0 hrs, 37 min	30	24
Problem 6	5 hrs, 59 min	0 hrs, 5 min	33	29

Table 5: RERS experiment results for AFL and KLEE

References

[1] "Rers challenge," http://rers-challenge.org/.

[2] "Klee," http://klee.github.io/.