# Reverse Engineering Report

Janina Roppelt *(s1194526)*, Joost Jansen *(s1370030)*, Ramon Houtsma *(s1245228)*

March 20, 2017

## 1 Introduction

Writing software can be compared to writing text. A script is a short story, a program might be more of a novel. When writing text little mistakes can occur and when writing a whole novel there might be logical errors. Checking long (self-written) text can be hard and errors might not be found due to tunnel vision on your own text. The same can be true for code. Fortunately code is a logic construct, which gives the possibility of automatic examination of code. One can extract knowledge or design information, which is called reverse engineering. This could afterwards be used to find input sequences of a program which can reach places in the program which shouldn't be reachable. To test and compare programs that do this analysis, reachability problems are created. Reachability problems are focused on complex structures to challenge programs. In the following two methods of reverse engineering are explained. After that two programs that use these methods are introduced. Finally these two programs are tested and compared with the help of reachability problems from 2017.

## 2 Fuzzing

Fuzzing is a technique that randomly generates new input values and then tries to find errors in the code with these values. A fuzzer works best within one 'compartment' of a code. This means that a fuzzer is good with code that has little logical gates[1]. Take for example a program that takes user input for an integer and expects 1, 2 or 3. If the switch case was not securely implemented a fuzzer could find security issues automatically by trying other values as input, because an integer variable can contain other values which may cause the program to crash.

There are two forms of fuzzing. One is blackbox fuzzing, the other is whitebox fuzzing. With blackbox fuzzing, the only thing known are well-formed program inputs. The fuzzer takes these inputs and modifies them randomly to try to trigger a bug or error like a buffer overflow (or a reachability error in the REAR cases). Whitebox fuzzing on the other hand has a little more depth. A program is executed symbolically which will enable gathering constraints on inputs from conditional branches. This overcomes the problem of a fuzzer, which mostly stays in one compartment of a software, and will sweep through many feasible execution paths[2].

## 3 Concolic execution

Concolic execution is a mix of CONCrete and symbOLIC execution of a program. It is used to find bugs in real-life applications rather than demonstrating logic program correctness. Symbolic execution is using symbolic expressions instead of concrete input data. With these constraints can be found and the paths through the program can be mapped. Think of a program with a lot of if statements. With randomly trying values it may take a long time to come through even one loop, while a concolic execution with symbolic constraints will be able to map all of them rather quickly.

There are a number of advantages of concolic execution. One is that external calls, for example to a library, can be performed with concrete values, while they would fail when using symbolic execution. The same also applies for code inside the program like for example unhandled operations or complex functions which cause constraint solver timeouts. These problems can be overcome by using concrete values, but will result in incomplete results. [3].

## 4 Descriptions of AFL and KLEE

### 4.1 AFL

American Fuzzy Lop[4] is a brute-force fuzzer coupled with an instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to pick up local-scale changes to program control flow. Edge coverage selects a test set T such that every branch of the control flow is exercised at least once by some d in T. The initial test cases in the queue are user supplied. AFL then attempts to trim the test case to the smallest size that doesn't alter the measured behavior of the program. Then it mutates the file using a variety of traditional fuzzing strategies, and if any of the generated mutations resulted in a new state transition it adds the mutates output as a new entry in the queue. The fuzzing engine of AFL uses algorithms to trigger unexpected behavior, which utilize methods such as bit flips, byte flips or simple arithmetics.

### 4.2 KLEE

KLEE is a *symbolic execution* tool that generates test cases based on bitcode, a form of white-box testing. [3] The basic idea of symbolic execution is that the input of the program is replaced with a symbolic variable $\lambda$. This symbolic variable is tracked while the program executes. At each statement KLEE branches into multiple paths, with each path containing a set of constrains called the *path condition.* The moment a path hits a bug, the path condition of the branch is used to find concrete inputs for a test case. These inputs can then be used for the raw version of the program, in order to generate the same bugs.

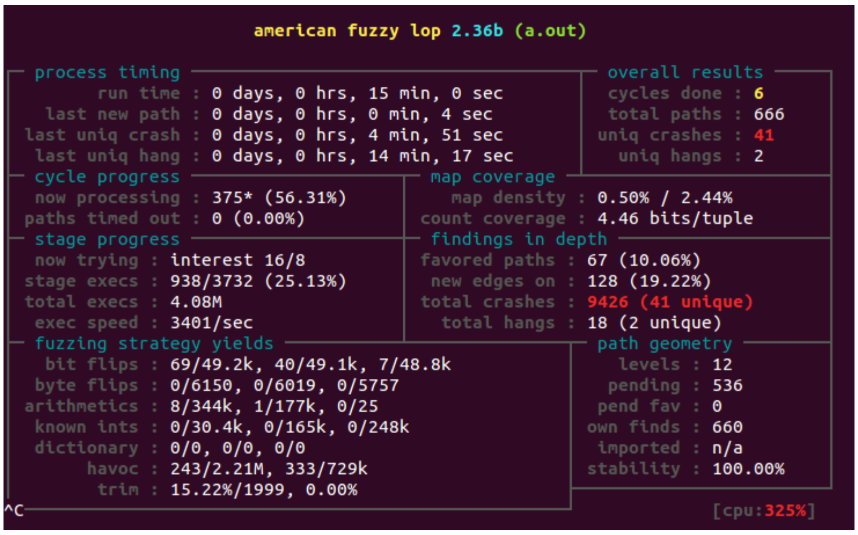
KLEE runs upon the LLVM-framework and creates its own symbolic environment. In addition, it uses the Simple Theorem Prover (STP) as a constraint solver. In KLEE, the symbolic variable is represented by formulas and the corresponding values for registers and memory locations are tracked throughout the execution. This ensures that each dangerous operation is checked. The fist stage is converting source code into LLVM-bitcode using the native compiler. Afterwards, KLEE is configured and the symbolic execution stage can begin. During this stage, KLEE follows the execution of the program and when no single path can be followed, the process is cloned and KLEE continues to follow both paths. Correspondingly, the path conditions of the branches are updated. When KLEE encounters a bug on a certain path, the path condition of that path is recorded and sent to the constraint solver. The constraint solver produces a set of constraints on the inputs, which lead the program to the bug. KLEE combines all the path conditions leading to bugs and converts them into test cases, along with metadata about the symbolic execution. To verify the bug independently of KLEE, the test cases can be applied to the real program.

A strong aspect of KLEE is that is is able to automatically generate high coverage test suites. It proved to perform better than manual test cases written by people. Another aspect is the fact that it can find deep bugs hidden in complex system programs, which have been undiscovered for a long time.

## 5 Experiments with AFL and KLEE on RERS problems

For the experiment problem 4 of the 2017 RERS problems was chosen first. For AFL the file was changed according to AFL specifications and compiled with afl-gcc. Five input files were created containing one integers ranging from 1 to 5. For KLEE, the files are changed to include KLEE and the KLEE run specification. The same is done for problem 5. For problem 6, five more input files needed to be created for AFL.

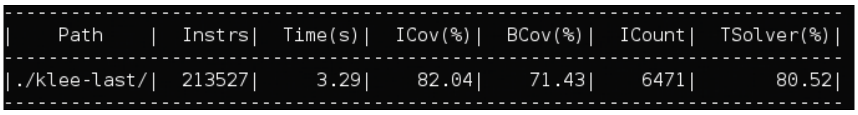
As both programs could be run for infinite time, it is chosen to let both programs run for fifteen minutes. In that way we tried to give them equal chances. Of course the speed of the program is also dependent on the computer it runs on. We did not account for that or any background programs running. Figure 1 shows the output of AFL after 15 minutes regarding problem 4. The output of KLEE regarding problem 4 is shown in Figure 2 for comparison.



```
                   american fuzzy lop 2.36b (a.out)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 15 min, 0 sec       │  cycles done : 6      │
│   last new path : 0 days, 0 hrs, 0 min, 4 sec        │  total paths : 666    │
│ last uniq crash : 0 days, 0 hrs, 4 min, 51 sec       │ uniq crashes : 41     │
│  last uniq hang : 0 days, 0 hrs, 14 min, 17 sec      │   uniq hangs : 2      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 375* (56.31%)     │    map density : 0.50% / 2.44%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.46 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 16/8         │ favored paths : 67 (10.06%)            │
│ stage execs : 938/3732 (25.13%)     │  new edges on : 128 (19.22%)           │
│ total execs : 4.08M                 │ total crashes : 9426 (41 unique)       │
│  exec speed : 3401/sec              │   total hangs : 18 (2 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 69/49.2k, 40/49.1k, 7/48.8k           │    levels : 12         │
│  byte flips : 0/6150, 0/6019, 0/5757                │   pending : 536        │
│ arithmetics : 8/344k, 1/177k, 0/25                  │  pend fav : 0          │
│  known ints : 0/30.4k, 0/165k, 0/248k               │ own finds : 660        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 243/2.21M, 333/729k                   │ stability : 100.00%    │
│        trim : 15.22%/1999, 0.00%                    ├────────────────────────┘
^C──────────────────────────────────────────────────────┘             [cpu:325%]
```

Figure 1: AFL result of problem 4



```
------------------------------------------------------------------------
|    Path    |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|./klee-last/|  213527|     3.29|    82.04|    71.43|    6471|       80.52|
------------------------------------------------------------------------
```

Figure 2: KLEE result of problem 4

## 6 Results

### 6.1 Produced outputs

**AFL** AFL produces three folders and three files as output. The three files contain firstly the fuzz bitmap, secondly the fuzzer statistics which include the data as displayed in Figure 1 and lastly plot data, which shows the progression of the fuzzing over time. The three folders are *crashes*, *hangs* and *queue*. The crashes folder contains unique test cases that led to a fatal signal, while the hangs folder contains unique test cases that caused the program to time out. The queue folder contains the test cases for every distinctive execution path. For the comparison with KLEE we are mostly interested in the test cases in the crashes folder. If we reproduce a test case we find to which error the test case led. The errors are collected using a self-written bash script, of which the code is available in Appendix 8. When we have reproduced all test cases we can compare the errors AFL found with the errors KLEE found in the RERS problems.

**KLEE** KLEE produces several files containing data and metadata about the run. Since finding the errors was the primary goal of this experiment, the errors that were found were written to a file (See appendix 8). Afterwards, the errors were filtered to remove duplicates. A list of unique errors was the final result. This method was applied to problem 4,5 and 6 of the 2017 RERS challenge.

The problem with KLEE regarding the RERS problems is that KLEE considers the assertion error as one unique error, while in fact it represents multiple errors. Due to this feature, KLEE only produces one .assert.err file with the inputs for the first error that is reached. KLEE also tells this via the command line: "KLEE: NOTE: now ignoring this error at this location". In order to find the trace to a specific error, one should add an extra IF statement to the code described in Appendix 8. Regarding the goal of this essay, this adaption was not performed.

### 6.2 RERS problem 4

**AFL** It found the last error just after ten minutes. The time difference between the last error and the one before that was around one minute.

**KLEE** At first, KLEE was ran using the command "klee -max-time 900 Problem4.bc". After 15 minutes, a lot of test cases were produced. While investigating the results, the suspicion arose that the errors could be found in a much shorter time. Again, KLEE was ran, this time using the command "klee -max-time 3 Problem4.bc". The number of found errors was equal to the 15-minute run.

Since all errors were found with both AFL and KLEE, these results are not very interesting for the purpose of this paper. The only conclusion we can draw was that KLEE was much faster than AFL in this case.

## 6.3 RERS problem 5

**AFL & KLEE** In two 15-minute runs on two different machines, the results were generated with both programs.

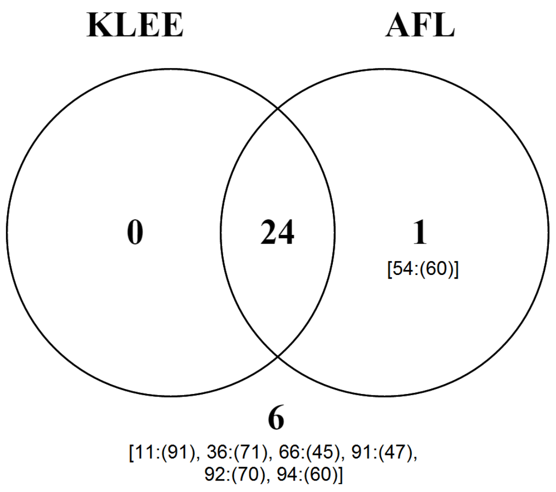


Figure 3: Venn diagram with the problem 5 results.

Figure 3 shows the results in a Venn diagram. The specific errors only one or no program found are given with: [Error1:(Complexity), Error2:(Complexity), ...], where complexity is defined as the number of input characters. Table 6.3 shows the complexity and coverage of the runs. Coverage is the percentage of possible errors reached.

| Problem 5 | AFL | KLEE |
|---|---|---|
| Coverage | 81% | 77% |
| Avg complexity | 8,84 | 6,71 |

Table 1: Problem 5 - Complexity & Coverage

## 6.4 RERS problem 6

**AFL** Two 15 minute runs were executed on two different machines, this time for AFL only.

**KLEE** Again, in a 15-minute run the results were generated. During the execution, KLEE displayed a "WARNING: killing x states (over memory cap)" twice.

| Problem 6 | AFL1 | AFL2 | KLEE |
|---|---|---|---|
| Coverage | 56% | 72% | 81% |
| Avg Complexity | 6,05 | 4,54 | 4,59 |

Table 2: Problem 6 - Complexity & Coverage

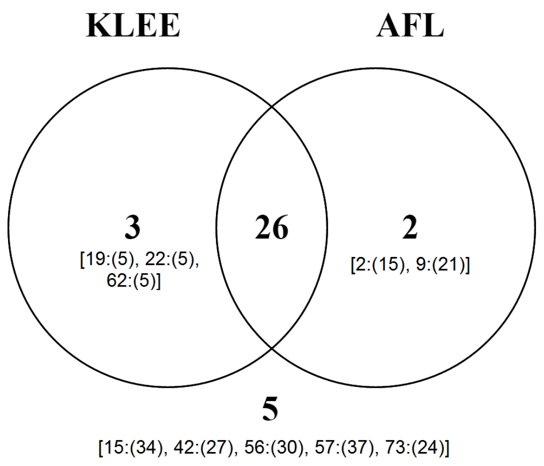


Figure 4: Venn diagram with the problem 6 results

Figure 4 shows the results in a Venn diagram. Again, the errors and corresponding complexity are displayed. The two runs of AFL are combined in the diagram. In Table 6.4 the Complexity and Coverage of the two runs of AFL and the run of KLEE are shown. Note that the complexity of AFL changes per run. Multiple runs with KLEE showed that the complexity stays the same between different runs.

## 7 Discussion

### 7.1 Errors found by only one program

**Problem 5** When looking at figure 3, it becomes clear that only error 54 was found by AFL and not by KLEE. This error is triggered by an input of 60 characters. This is remarkable, since there are three errors with less or similar complexity that haven't been found. There is no explanation why this error has been found and the other ones haven't; this is probably the result of the randomness in mutating the test cases of AFL. The disadvantages of KLEE become clear during this run; the growing number of inputs result in an exponential increase of branches. [3]. KLEE delivers high coverage test suites for simple programs, increasing the complexity for the errors results in a huge amount of branches. AFL is not restricted by this feature, thus having a better chance of finding complex errors.

**Problem 6** Several interesting findings are shown in figure 4. Again, AFL found two errors that were undiscovered with KLEE. However, this time KLEE found three errors that were undiscovered by AFL. These three errors have a complexity of 5. This confirms the high-coverage reputation of KLEE regarding "simple" programs[3]. Furthermore, five errors haven't been found at all. These undiscovered errors have a high complexity, making it very hard for AFL to find it and difficult for KLEE to reach the correct branch.

### 7.2 Reflection on usability

With both AFL and KLEE the code has to be modified in order to work with the programs. In AFL the input has to be specified in separate files as well as in the code. Forgetting, or not completing this step will lead to significantly less results. The user interface of AFL while running is more fun. While KLEE only prints the outputs and you see some errors flying by, AFL has a nice interface that tells you variables like when the last crash occurs and how many crashes are there. KLEE on the other hand has more run options, for example a specification for how long the program should run. Unfortunately this did not seem to be very precise. A problem with both programs is that errors are classified as separate errors more than once, probably because of a different path leading there. This also leads to the results not being clear without further analysis.

## 8 Conclusion

Based on the literature and the findings in this essay, some statements can be made. First of all, KLEE has a high coverage on the lower complexity problems and a low coverage on high complexity problems. This is a direct result of the symbolic execution, which causes KLEE to expand exponentially when the complexity increases. Secondly, AFL has found some high complexity problems, but these results are hard to reproduce since they are the result of randomness. The limiting factors for AFL and KLEE are respectively time and memory. For the best coverage, a combination of a fuzzer and a concolic execution should be used. As the fuzzer is strong in one compartment of the code and a concolic executor on finding new compartments they complement each other. Programs like this already exist, an example is Driller [1].

## References

[1] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, and year=2016 Christopher Krügel and Giovanni Vigna, booktitle=NDSS. Driller: Augmenting fuzzing through selective symbolic execution.

[2] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

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

## Appendices

### Bash script for collecting AFL errors

```
#!/bin/bash
for i in `seq 0 9`;
do
        cat output/crashes/id:00000$i* | ./a.out 2>> tmp.txt
done
for i in `seq 10 99`;
do
        cat output/crashes/id:0000$i* | ./a.out 2>> tmp.txt
done
while read line; do
        if [[ "$line" == *"Invalid"* ]]; then
                :
        else
                echo $line >> errors_unsorted.txt
        fi
done <tmp.txt
sort -u errors_unsorted.txt > errors.txt
rm tmp.txt
```

### Adapted C code for collecting KLEE errors

```
void __VERIFIER_error(int i) {
        /*Add this to get trace to specific error
        *if(i == ERRORNR) { */
        fprintf(stderr, "error_%d \n", i);
        FILE *f = fopen("output.txt", "a");
        if (f == NULL)
        {
        printf("Error accessing file");
        exit(1);
        }
        fprintf(f, "error_%d \n", i);
        fclose(f);
        assert(0);
        /* } */
}
```

Afterwards, in bash;

```
sort -u output.txt > sortedoutput.txt
```