The differences between fuzzing & concolic execution

Daan Schipper - 4155270
Ruben Starmans - 4141792

March 20, 2017

1 Introduction

This report describes the differences between fuzzing and concolic execution. First the two terms will briefly be explained and an example for each will be given in Section 2 and 3. Next, two tools, AFL[1] and KLEE[2], will be detailed that make use of the techniques in Section 4. Section 5 describes the experiments run on reachability problems[3]. Lastly, in Section 6 an analysis will be given of the performed experiments.

2 Fuzzing

Fuzzing is a form of random testing which mutates existing program inputs and then test the program with those new inputs. The goal of this form of testing is to trigger bugs such as buffer overflows. This method is known as blackbox fuzzing and can be very effective. However, because of the random generation of new inputs for the code it can be very slow in finding bugs and issues. The following small piece of code only has a 1 in 2^{32} chance of being executed with x being the right value when it is a randomly chosen 32-bits value.

Listing 1: Basic code with if statement

```
int foo(int x) { // x is an input
  int y = x + 3;
```

[1]http://lcamtuf.coredump.cx/afl/
[2]http://klee.github.io/
[3]http://rp16.cs.aau.dk/

```
  if (y == 13)
    abort(); // error
  return 0;
}
```

Whitebox fuzzing was later introduced and was aimed to improve on the issues blackbox fuzzing has. Whitebox fuzzing no longer keeps randomly testing the program with randomly mutated inputs. It rather generates one new input and tests the program with that input. It then continues to adapt the input but it does that by negating constraints systematically, solving those with a constraint solver and then mapping that to the input. Executing whitebox fuzzing on the piece of code above would run with a value 0 for x and would not pass the if statement. Then the constraint would be negated and it would give an input value of 10 for x. This drastically improves on the time needed to fuzz the code and find most of the different cases based on the constraints [1].

3 Concolic Execution

Concolic execution is a combination of concrete execution and symbolic execution. With symbolic execution, all possible path conditions are found in a program. A path condition is a set of logical constraints to reach a specific point in the execution. This is then combined with concrete execution, concrete inputs are generated to test the program with the aim of maximising test coverage.

Listing 2: Function *get_sign*

```
int get_sign(int x) {
  if (x == 0)
    return 0;
  if (x < 0)
    return -1;
  else
    return 1;
}
```

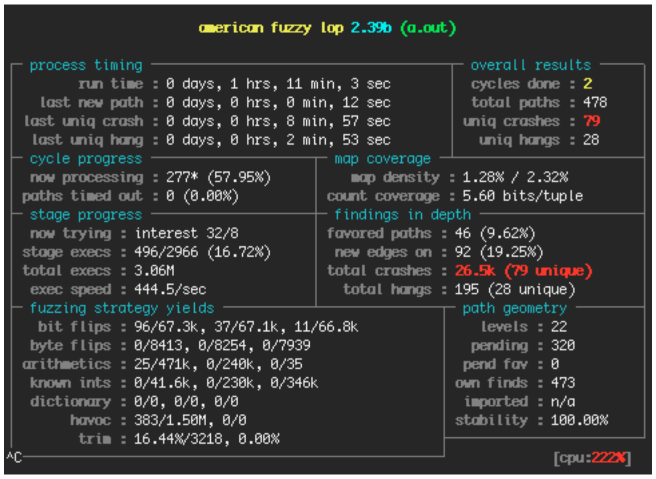
The function *get_sign* will be tested with a symbolic variable to find the three path conditions present in the function, one where x is 0, one where x is less than 0 and one where x is greater than 0. With concolic execution three test cases will be generated for the function with each a different value of x, for example 0, -4103 and 1239876, to exercise all paths.

4 Description of AFL & KLEE

In this section both testing tools will be briefly describer. Both tools will be explained how they work and how they should be used.

4.1 AFL

American Fuzzy Lop (AFL) is a security oriented fuzzing tool. It uses a genetic algorithm to find clean and interesting test cases. This improves the functional coverage for the fuzzed code.



```
                      american fuzzy lop 2.39b (a.out)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 1 hrs, 11 min, 3 sec       │  cycles done : 2     │
│   last new path : 0 days, 0 hrs, 0 min, 12 sec       │  total paths : 478   │
│ last uniq crash : 0 days, 0 hrs, 8 min, 57 sec       │ uniq crashes : 79    │
│  last uniq hang : 0 days, 0 hrs, 2 min, 53 sec       │   uniq hangs : 28    │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 277* (57.95%)     │    map density : 1.28% / 2.32%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 5.60 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : interest 32/8         │ favored paths : 46 (9.62%)            │
│ stage execs : 496/2966 (16.72%)     │  new edges on : 92 (19.25%)           │
│ total execs : 3.06M                 │ total crashes : 26.5k (79 unique)     │
│  exec speed : 444.5/sec             │   total hangs : 195 (28 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 96/67.3k, 37/67.1k, 11/66.8k          │    levels : 22         │
│  byte flips : 0/8413, 0/8254, 0/7939                │   pending : 320        │
│ arithmetics : 25/471k, 0/240k, 0/35                 │  pend fav : 0          │
│  known ints : 0/41.6k, 0/230k, 0/346k               │ own finds : 473        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 383/1.50M, 0/0                        │ stability : 100.00%    │
│        trim : 16.44%/3218, 0.00%                    ├────────────────────────┘
^C────────────────────────────────────────────────────┘             [cpu:222%]
```

Figure 1: Statistics that AFL gives while fuzzing

AFL requires almost no configuration. As input it needs compiled code that has to be tested and specified paths to test cases and the path where the tool should put its findings. It then starts fuzzing and gives a screen with statistics which is shown in Figure 1. Information such as running time and current progress is shown. But also results with respect to how many unique crashes it has found for the input code.

What AFL is best at is maximisation of path coverage which it does in minimal time with its method of structurally negating constraints. The biggest limitation of AFL is that it is currently optimised only for binary files.

4.2 KLEE

KLEE is a symbolic execution tool, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs.

To test the function in Listing 2 with KLEE some adjustments have to be made. In order to let KLEE know that variable a is symbolic, the line $klee\_make\_symbolic(\&a, sizeof(a), "a")$ is added, as seen in Listing 3.

Listing 3: Function main

```
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

The code has than be compiled to LLVM bitcode, on which KLEE can operate.

KLEE does not currently support symbolic floating point, longjmp, threads, and assembly code. Additionally, memory objects are required to have concrete sizes.

5 Experiments on AFL & KLEE

In this section the experiments that were executed with AFL and KLEE on the Reachability problems will be described. In section 6 these experiments will be further analysed.

5.1 Output AFL

For AFL the problems 10, 11 and 12 were run for at least 50 minutes each. Problem 11 was run for a bit longer than that in order to complete at least two cycles. The later problems were not run for that long because they completed cycles so quickly and because in any of those cycles no crashes were found. So the focus is mainly on the problems ten and eleven because most crashes were found there. In Table 1 the results of the fuzzing of the problems can be seen.

5.2 Output KLEE

Each problem of the reachability problems has been run with KLEE with the the setting max-time of 600 seconds (10 minutes). The results can be

Table 1: AFL Statistics

Problem	**Time (m)**	**Instr (Millions)**	**#Cycles**	**Unique Crashes**
10	54	2.51	15	44
11	71	3.06	2	79
12	53	2.30	32	0
13	4	0.16	75	0
14	7	0.38	162	0
15	13	0.65	78	0
16	25	1.08	14	0
17	14	0.54	99	0
18	14	0.73	422	0

found in Table 2.

As seen in the table, for some problems KLEE did not produce a noticeable result. KLEE can be run with different search heuristics (dfs/bfs/random-state/random-path/nurs:covnew/nurs:md2u/nurs:icnt/nurs:cpicnt/nurs:qc) and different backend solver (MetaSMT, Z3), but this resulted in exactly the same result.

Table 2: KLEE completed paths and generated tests

Problem	**Total instructions**	**Completed paths**	**Generated tests**
10	43061863	56620	26967
11	48752343	52690	46917
12	72741	54	29
13	1896	2	2
14	2322	2	2
15	7793	6	4
16	82010	47	25
17	8306	4	3
18	1501	2	2

6 Analysis

In this section the outputs of the experiments executed in section 5 will be analysed. The difference in the errors found by the respective tools will be explained and lastly the future usability of the tools will be evaluated.

Table 3: Klee stats

Problem	**Instr**	**Time (s)**	**ICov (%)**	**BCov(%)**	**ICount**	**TSolver (%)**
10	43061863	785.32	90.10	71.55	4877	9.51
11	48752343	987.16	87.69	65.99	15182	6.69
12	72741	2.46	3.86	4.41	75219	83.12
13	1896	0.18	0.84	2.64	224925	50.66
14	2322	0.16	1.12	3.05	206971	42.02
15	7793	0.65	0.53	1.06	562422	74.57
16	82010	10.35	0.37	1.28	1262570	93.35
17	8306	28.98	0.09	0.86	4214015	4.62
18	1501	1.16	0.07	0.25	2067320	19.24

6.1 Produced Outputs

6.1.1 AFL

Output generated by AFL is put into a findings folder. In this folder some statistics about the fuzzing that took place can be found. But also two sub folder names crashes and hangs. The folder containing the crashes is the most interesting and will be discussed here.

In the crashes folder there is a file create for each unique crash that happened. Each file has a name that looks something like this: id/000000,sig/06,src/000001,op/havoc,rep/4. In this name some information can be found such as the crash id or what operation (op/) the fuzzing was performing. When this name is run in the terminal with the following command the error that the crash reaches is printed.

Listing 4: Command to extract output from the generated files

```
cat id:000000,sig:06,src:000001,op:havoc,rep:4 |
    path_to_binary/a.out
```

6.1.2 KLEE

KLEE produces for each run global files, of which the ones used for this report are follows:

1. **info**: This is a text file containing various information related to a KLEE run. In particular, it records the exact command-line with which KLEE was run, and the total time taken by the execution, as seen in Table 2.

2. **run.stats**: This is a text file containing various statistics emitted by KLEE. The results can be seen in Table 3.

The following files are generated per path condition:

1. **test<N>.ktest**: Contains the test case generated by KLEE on that path.
2. **test<N>.<error-type>.err**: Generated for paths where KLEE found an error. Contains information about the error in textual form.
3. **test<N>.kquery**: Contains the constraints associated with the given path, in KQuery format.

6.2 Errors Found

While analysing the produced outputs we found that there were a lot of errors that both tools had found. However for problem 10, AFL has found 31 unique crashes and KLEE has found 44. So there are definitely some differences. Below we will discuss on errors that AFL found and KLEE did not and vice versa. Error 16 was found by KLEE and not by AFL and vice versa for error 99. This is probably because of the way both tools work. Error 99 needs a very complex input for it to trigger which is why AFL with its method of negating constraints one by one could find it. KLEE probably could not trigger this error because it has a limitation to how deep the if statements are checked.

Why error 16 was found by KLEE and not by AFL is also because of the difference in how they work. The if structure was not complex enough for KLEE to not find it and AFL probably did not find it because it could not negate all the constraints in the right order.

6.3 Future Usability

Both tools have a promising future ahead because more and more software is written everyday which needs extensive bug hunting for it to be stable enough. However while they are both useful tools for findings bugs for extreme and unusual test cases they might need some work to be fit for the future. The way we write software is ever changing and thus the way we could test them could also change and improve with that. The biggest issue for AFL which need to be addressed for future use is to improve the tool to accept other inputs than binaries. For KLEE the largest hurdle to overcome is the trouble it has with large scale project.

References

[1] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.