CS5218: Assignment 4

Comparison of the AFL and KLEE

Deadline: 23:59 Sunday 21st April 2024

We consider the problem of *bug finding*, and in particular, how *path coverage* addresses this. You are expected to experiment with *fuzzing, via AFL*, and *symbolic execution, via KLEE*[1]. You are to articulate some thoughts, via *specific concrete examples* on the efficacy of these two systems. In particular, you are to *compare* them against one another.

You are to organize yourselves into *three-person teams* (with an exception of one size 4 team). Please indicate the team structure for this project in the shared Google Spreadsheet. There are a total of 15 unique program sets. Each set consisting of 10 programs available from the appropriate folder on Canvas.[2].

An Academic Program

Consider the pseudocode below, where `input` is to be provided by the system (AFL or KLEE) and `output` is conditioned by an unspecified predicate β which, if true, indicates that a *bug* is encountered.

```
#define N ...  // ..............................  problem size
unsigned long int input, output = 0;
int k = 0; // ...................................  branch encounter
int tmp = input % 2^N;
for (int i = 1; i <= N; i++) {
  if (tmp % 2) output += 2^k;
  k++;
  tmp /= 2;
}
if (β(output)) assert(0); // ......................  BUG!
```

Note that the (constant) parameter N defines the size of the problem instance. It is a number that is large, but far smaller than the total number of possible inputs. This program is

[1] AFL and KLEE installation documents are available on Canvas

[2] For the four-member team, we will provide three extra programs

constructed to have exactly 2^N paths, each with a *unique* output. Therefore, by varying the predicate β, one can make bug detection as easy (eg. $\beta(x) \equiv (x \geq 0)$) or as difficult (eg. $\beta(x) \equiv (x == \kappa \text{ for some } 0 \leq \kappa < 2^N)$) as desired.

Task 1: Fuzzing with AFL

Assume the input values are uniformly distributed, in the range $[0, 2^{64} - 1]$. The output range is $[0, 2^N - 1]$ for some N generally much smaller than 64. It is easy to see that the output value is also uniformly distributed (ie each output value is equally likely).

As mentioned above, bug finding here means to define a *target* predicate on the final output value. For example, a target of $output \leq 2^N - 1$ would result in *every* input value delivering a discoverable bug. On the other hand, a target of $output == \kappa$, for some fixed constant κ, would mean that each input value would only hit the target with a small $(1/2^N)$ probability.

Your task is to concretize the pseudocode program into a C program that can be run by AFL. Your experiments will be limited by some *timeout*.

In your C program, you will need to have some *instrumentation* that *persistently records* the path on each AFL run so as to finally compute the total number of *distinct* paths executed during execution over a specified timeout period. (Hint: write to a file and process that file at the end.) Because we know the actual total number of paths, you can then report *path coverage* for your experiment. Recall that AFL pre-requires a seed (or multiples thereof).

- SEEDS, PREDICATES
  Perform your experiment reporting on varying seeds and bug predicates (β).
- COMPARE WITH RANDOM INPUTS
  Now perform your experiment not with AFL, but with using *random* inputs (over the input range $[0, 2^{64}]$). You will need to use the standard `rand()` function of C which follows a uniform distribution.

Present all your results and thoughts.

Task 2: Fuzzing with KLEE

Our academic program has a finite number of paths, and so symbolic execution, via KLEE for example, will be able to examine *all* the paths. However, the *throughput* of KLEE, ie. the

speed of processing each path, is clearly substantially slower than that of fuzzing as in AFL. (By how much?) This is partly due to the fact that KLEE is an *interpreter* as opposed to having *native execution* as in AFL, and additionally that KLEE depends on a *constraint solver* (such as an SMT solver) in the process of path consideration.

In this task, you are to experiment KLEE to investigate its coverage relative to its given timeout. (Note that you need to declare that the variable `input` is *symbolic*, and that there is no need for any seeds.)

In particular, you are to demonstrate some circumstances when the higher throughput of AFL is ultimately superior over KLEE, and you are to demonstrate the converse conclusion.

In the end, present a narrative on the comparison.

Task 3: Investigate given Programs

In this task, you have to analyze a provided set of programs (10 programs); and compare AFL and KLEE with variances on the AFL seed(s), the problem size and the timeout. These programs may contain not one bug, but several. If so, you may also compare the systems for the *number of bugs* detected.

Also, you will need to instrument the programs to track the number of distinct paths encountered. This will be a manual process.

In general, the total number of paths for a program is not available, unlike in our academic example. You will thus assume that these programs are run in a *bounded* setting so that every path is finite. One way to do this is to bound the loops. Since you can control the *level* of the bounds, you will be able to set a level where KLEE can perform *complete search*. In this case, you can report the exact *path coverage* of AFL (while KLEE's is of course 100%). In other cases, where you do not have the total number of paths, you can just compare the number of paths encountered by AFL vs KLEE.

Suggested Command-line Arguments for Each Tool

Fuzzing - AFL

AFL can be instrumented and ran with the following commands:

1. Instrument: afl-gcc -fno-stack-protector -z execstack test1.c -o test1
2. Run: afl-fuzz -i ./testcase/ -o ./results/ ./test1

Important options are as follows:

1. -i dir –input directory with test cases
2. -o dir – output directory for fuzzer findings
3. -t msec – timeout for each run (auto-scaled, 50-1000 ms)
4. -m megs – memory limit for child process (50 MB)
5. -M / -S id – distributed mode (see parallel_fuzzing.txt)
6. -C – crash exploration mode
7. timeout 600 – Prefix this option to command line for killing the process

For more detail please run *afl-fuzz –help*.

Symbolic Execution - KLEE

KLEE can be run with different search strategies and command-line arguments. You can choose different search strategies with different memory/time bound:

1. –search=dfs: Runs KLEE with the DFS search strategy.
2. –search=random-state: Runs KLEE with the maximum coverage search strategy.
3. Other search strategies can be seen by running "klee –help".
4. The command-line options "–max-memory=5000" and "–max-time=600" force KLEE to run on at most 5000 MB of Ram and for at most 10 minutes.

Submission

A 5-10 page report in PDF on your findings where you can emphasize your evaluation of the systems (`project.pdf`). It should describe your experiment setting, results and some discussion on which strategy you found more useful. Please also add the stats for the tested programs. This report is to be submitted by 23:59 Sunday 21 April 2024, ie. at the end of *WEEK 13*.

Make sure your submission contains information about your name, matric number, and email. Your zip files should have the format *Surname-Matric*-asg4.zip. Submit all files above to the appropriate Canvas Submission Link.