AISTR Lab 2

Marios Marinos - 5353106
Mihai Macarie - 4668634

March 12, 2021

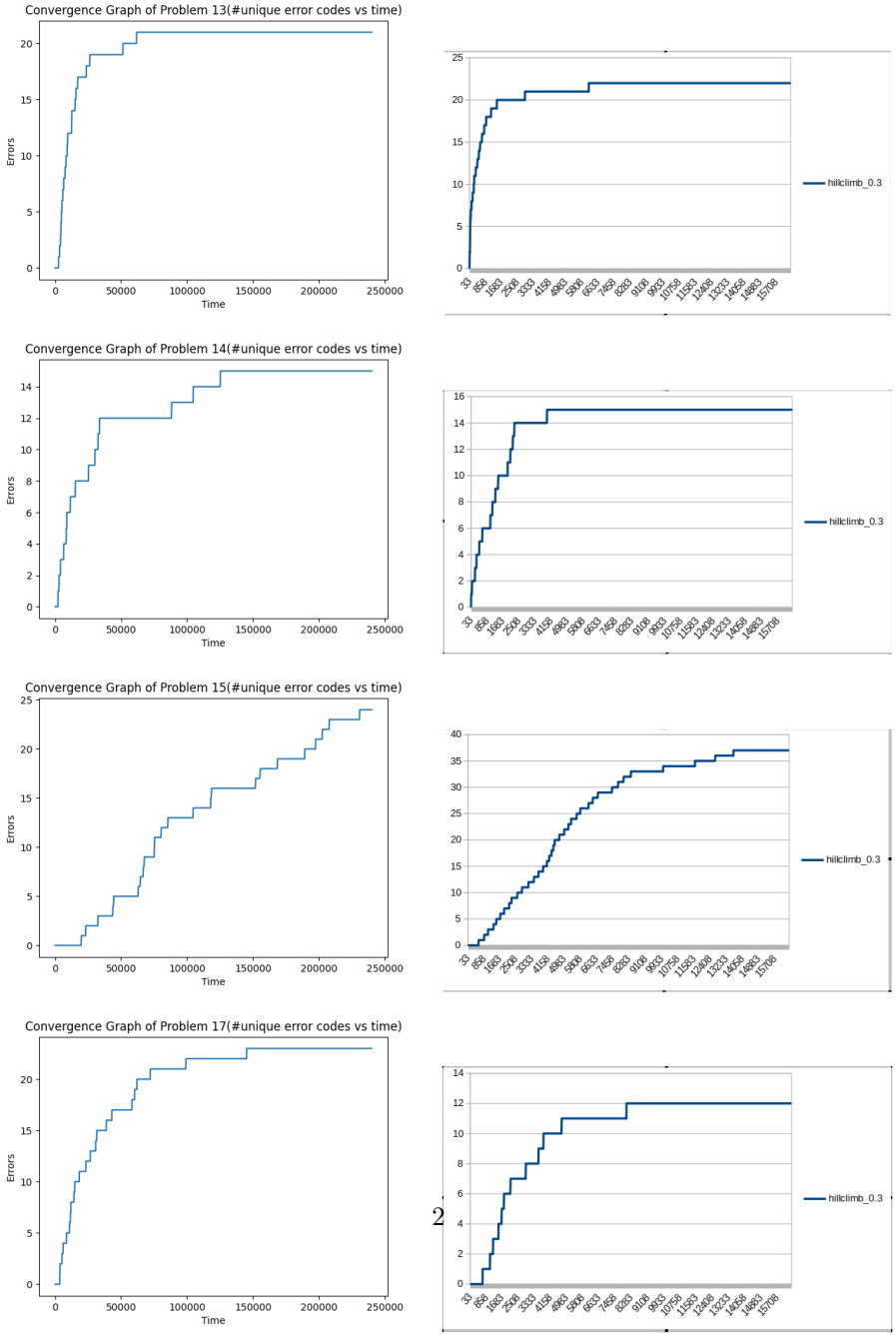
1 Task 1: symbolic/concolic execution

The Symbolic execution that was implemented in our code, follows pretty much the idea that is proposed in the assignment. Firstly, the fuzzer generates a random trace of Length that you can set in the beginning (Trace Length used for the graphs was 100). Then, it starts solving the opposite branch and save that in a queue, in case the condition is satisfiable, that will be used later to detect the new branch. A small "trick" that I used that helps, especially at the start of the fuzzer is that I add 5 random symbols after I get the solution from the solver, to explore a bit more. Also, I keep a hashSet to keep track of which branches are covered yet, and in case they have been already solved I just skip those branches to save time. This is pretty much the implementation of the symbolic execution. To make the graphs, I used a .csv file that is given by the fuzzer, and then I use a python script to output the graphs shown in Figure .

Figure 1 is shown the convergence graph of the hill climber fuzzer used in Lab 1 and the symbolic execution + fuzzing in Lab 2. On the left Figure 1 column are the convergence graphs using symbolic execution from 13 to 17 (without Problem 16). Because on Figure 1 is not easy to see how many errors are in the numbers, I also provide the final unique errors in the table 1. The Symbolic execution fuzzer was running for each Problem for 5 minutes, whereas the hill climber did run a bit less (around 2 min) but it's also quite obvious that the hill climber converges faster than symbolic execution. I do think, that between the 2 strategies there is not one that is better or not. It depends on the problem that we are facing off, as we can see that for some problems Hill climber is doing better(e.g. Problem 15), whereas the Symbolic fuzzer does better for different problems (Problem 17). In the long run, intuitively, I would say that Symbolic execution would do a bit better given the right trace length.

	Hill Climber(Unique Errors)	Symbolic Execution(Unique Errors)
Problem 13	22	21
Problem 14	15	15
Problem 15	37	24
Problem 17	18	23

Figure 1: Number of found unique errors on the vertical axis, against the milliseconds on the horizontal axis, for problems 13,14,15,17 from left to right.



2 Task 2: KLEE vs AFL

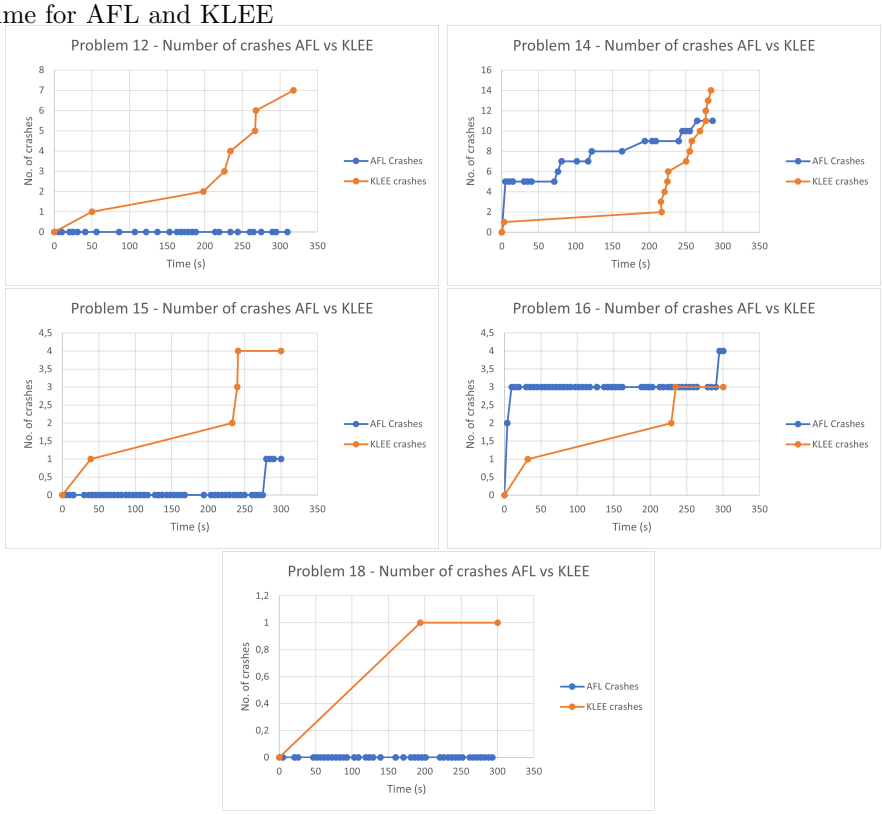
AFL is a random genetic fuzzer capable of synthesizing complex file semantics guided by instrumentation, optimized for edge coverage. It works by loading inputs defined by the user in a queue, taking the first input, trimming the current test case as much as possible it does not change the behavior, mutating the file using various fuzzing techniques, and if a new state transition is found, it is added to the queue. Then, the next input from the queue is taken. AFL is good at testing user-defined inputs and finding bugs.[1, 2]

KLEE is a dynamic symbolic execution engine base on LLVM infrastructure. It works by using symbolic inputs and compiling the code to LLVM bitcode. Compared to AFL, it does not use user-specified inputs. It is good at high line coverage and research shows it discovered errors that were not found using other methods.[3, 4]

AFL and KLEE were run last day in Windows environment, and that's why we think the results are quite bad compared to our results in first lab(for AFL). Windows are much slower compared to Unix-environment and thus, it did only manage to find little errors. Unfortunately we didn't have enough time to fix this.

RERS Problems that we used were 12,14,15,16, and 18. We ran AFL and KLEE for 5 minutes (300 seconds) for each of these problems. In Fig. 2, we can notice that KLEE has discovered more crashes than AFL, except for Problem 16. KLEE have found more errors, since we let the tools run for a little time (meaning that they didn't manage to reach enough paths in this time) and symbolic execution would be better when it comes to that, whereas hill climber and AFL would need a bit more time to reach more errors.

Figure 2: Number of found unique errors(crashes) on the vertical axis, against time for AFL and KLEE



References

[1] american fuzzy lop. `https://lcamtuf.coredump.cx/afl/`. (Accessed on 03/10/2021).

[2] Github - google/afl: american fuzzy lop - a security-oriented fuzzer. `https://github.com/google/AFL#:~:text=In%20this%20mode%2C%20the%20fuzzer,it%20in%20the%20crashing%20state.` (Accessed on 03/10/2021).

[3] Klee. `https://klee.github.io/`. (Accessed on 03/10/2021).

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, Dec 2008.