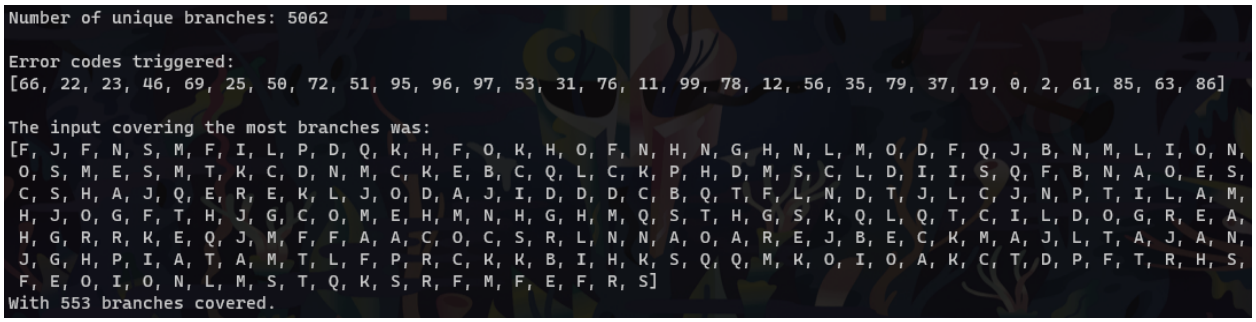
CS4110 AISTR Final Lab Report

Group 60

1 Lab Recaps

1.1 Lab 1 - Fuzzing

For the fuzzing lab the main technique is a simple discrete hill climber algorithm. An input trace is initialized randomly from a set of valid symbols, and all subsequent traces are also composed of these symbols. The primary loop of the algorithm keeps track of the best trace, which is found by running a program with those inputs, keeping track of whenever a new (conditional) branch is found and calculating the branch distance from the encountered branch to a target branch. In this case the target branch is its sibling/negation. Those branch distances are summed up, and this total branch distance is used in the hill climber when optimizing for branch coverage, which works by randomly mutating the previous trace a number of times and comparing total branch distances to the previously found best. If no new best trace is found after this process (which is what we call fuzzing), then the length of the trace is extended and re-initialized randomly. The extension allows for deeper branches to be reached, which is what we want when trying to induce errors.
Below is an example of the output after running the algorithm for 5 minutes on problem 17:



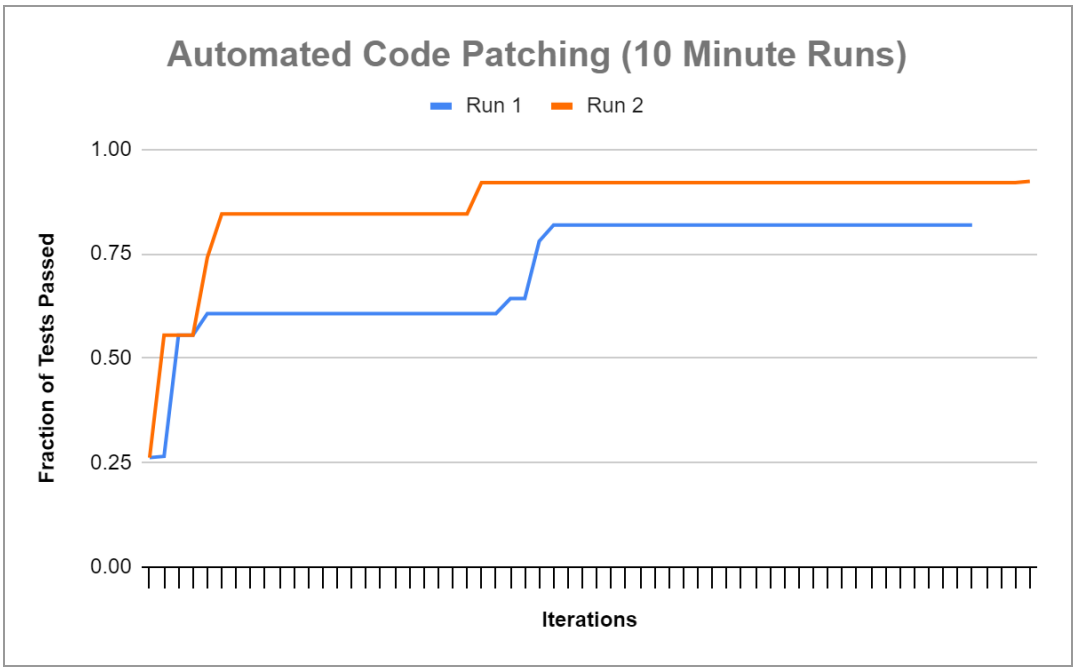
As mentioned, we can see that the input covering the most branches is fairly long, which is because extending the trace leads to discovering deeper branches. An important part of the design of the hill climber is what type of and how many mutations to make to the input trace when doing the *local search* (repeated random mutations and comparison). In an earlier design we only mutated a single element in the trace, which turned out to be too little. Adding more mutations improved the performance of the algorithm greatly. This kind of tuning is problem dependent however, and the shape of the optimization landscape will ultimately determine what kind of mutations/search radius gives the best performance under time constraints.

1.2 Lab 2 - Symbolic Execution

Text

1.3 Lab 3 - Automated Code Patching

This lab focuses on an implementation of a genetic algorithm to create patches to a set of buggy programs. Limiting the scope of the task, and through code instrumentation, the problems are effectively parameterized as an ordered set of operators that correspond to the operators found in conditional statements in the code (==, >, !=, etc.). Taking inspiration from the evolutionary process present in biological beings on earth, the algorithm loops over 4 main subroutines: evaluation, selection, crossover and mutation. These routines act on a population of individuals, which in this case is the collection of all the sets of operators, representing the patched programs, at any given iteration of the loop. Each set of operators is evaluated by running a test suite on the program with those operators inserted, giving us a fitness defined in terms of the number of tests passed and failed. Simply using the fraction of passed tests to total tests was deemed sufficient for the purposes of the lab. Fault localization is also performed in this step, via the tarantula score. Next up a tournament-style selection is performed, where 2 individuals are picked at random and compared based on their previously evaluated fitness score, removing the loser from the population. This continues until we are left with a certain fraction of the population remaining, defined as a hyperparameter of the algorithm. Following the selection, we reproduce (create new individuals) through a single point crossover method, combining operator sets from the remaining individuals, again picking 2 at random. This is done until the population once again reaches its initial size, which is another hyperparameter. Finally we randomly mutate a small percentage of the population, also defined by a hyperparameter in terms of a probability to mutate any given individual. The mutations are done by selecting a random operator, and if it is sufficiently suspicious, defined by a threshold tarantula score, change it by random selection from a list of valid operators.



The figure above shows how the highest fitness score in the population evolves over time for a 10 minute run on one of the RERS problems (problem 12). For problems where there are a lot of changes to be made, the crossover part of the algorithm contributes the most during the early and middle portions of the runtime, but to be able to reach the best possible fitness score of 1.0, the random mutations are crucial. In our implementation, mutation can occur to any individual with equal probability, so there is no guarantee that the best individual is carried over to the next generation, but it is highly unlikely that all the best individuals are mutated. We can see that for two 10 minute runs of the algorithm the best individual was carried over at each iteration. Inherent in the algorithm is also a sizable element of stochasticity, and especially for short runtimes we see a meaningful variation in the performance just from this.

1.4 Lab 4 - Model Learning

Text

2 AFL

2.1 Comparison: AFL, Smart Fuzzer, Random Fuzzer

Example of reference to a figure using link to heading. Can also use a link to bookmark. Both work with *Download → PDF Document* and *Print → Save to PDF*, but **not** *Print → Microsoft Print to PDF*.

3 KLEE vs AFL

3.1 How They Work

Text

3.2 Comparison: KLEE, AFL

Text

3.3 Analysis of Results

Text

4 Genetic Programming using ASTOR

4.1 Analysis of Generated Patches

Running the ASTOR tool on buggy RERS problems 1-3, 7, 11-14 and 17 for a max time of 5 minutes and a max number of iterations/generations set to 3000, yields the following results:

Problem	1	2	3	7	11	12	13	14	17
Patches	21	0	0	0	38	0	2	0	0
Time	5 min	D.N.F.	D.N.F.	D.N.F	5 min	D.N.F.	5 min	D.N.F.	D.N.F.

As can be seen from the table, many of the problems do not finish (or even time out) but get stuck for a long time.[1] Comparisons of the generated patches to the buggy and non-buggy versions of the problems reveal that ASTOR indeed finds the correct patch, as in it replaces the correct operator at the correct line, for problems 1 and 11, but not for 13. Keeping in mind the time constraints enforced, it is possible that longer runtimes would result in patches (correct or otherwise) for more of the problems. With that said however, of the 21 patches supplied for problem 1 for instance, only one or a small portion of them are correct when compared to the ground truth, and as such we are ultimately limited by our test suite which determines what patches ASTOR will accept. As far as patching so that all the tests pass, the patches are meaningful, but finding the correct patch to deploy when multiple are returned, that all pass the

[1] This issue was also discussed on Mattermost, and it was deemed too time-costly and of little pedagogical value to run ASTOR for much longer than 5 minutes, because others have reported no change even if ran for as much as 45 hours.

tests, is not easy. This is likely mitigated when applied to non-obfuscated code where one can qualitatively review the suggested patches. Finally, the returned patches also contain many logically equivalent patches, which obviously are not all meaningful in the sense that they change the program's logical behavior in the exact same way.

5 LearnLib

5.1 Comparison: LearnLib, Students' Implementation

Text

Visualizations

1 Placeholder Image