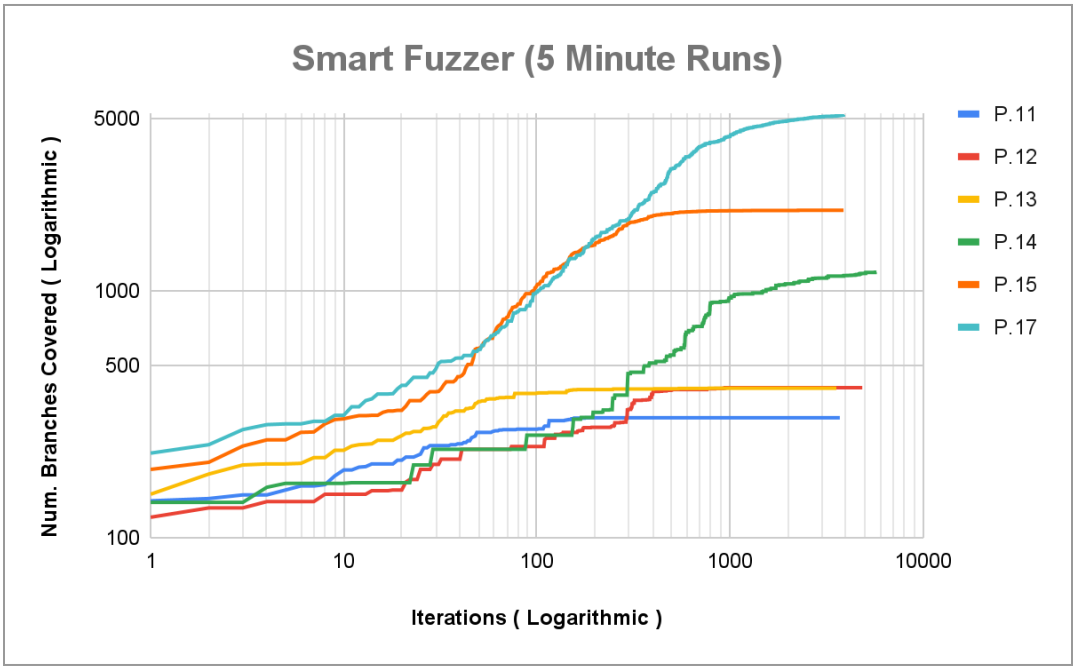
CS4110 AISTR Final Lab Report

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.



From the figure above we can see that for all the problems the branch coverage increases exponentially (linear on log-log plot) in some regions and then flattens out. The flattening is either because all branches have been covered, in which case we obviously can't increase branch coverage, or when the remaining branches are very deep. After running the algorithm we observe that the input covering the most branches (not shown here) is fairly long, which is because extending the trace leads to discovering deeper branches.

Problem	11	12	13	14	15	17
Num. Branches Covered	306	405	403	1189	2118	5160
Num. Errors Triggered	18	15	22	15	40	30

Results Hill Climbing Fuzzer

Some performance statistics from the runs are shown above, and we note that by looking at the convergence of the algorithm in the graph, we can be more confident that we reached full branch coverage for problems 11, 12, 13 and 15, than for 14 and 17. One possible improvement could then be to implement a smarter scheduling of trace length, to allow for the possibility of finding very deep branches after the algorithm has seemingly converged. Another 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

With Symbolic Execution the key idea is to use a SAT solver to check if a branch can be reached or not. The algorithm has a number of states in its queue which starts with only the program in its initial state. A state here means all the information (variables, current line number, etc) that differentiates that can be used to differentiate states. It then takes a state from its queue, symbolically executes one line of code by changing the state information and repeats that until the queue is empty.
When processing a line a number of things can happen that can add states to the queue, the main one being branches. If a branch is encountered the SAT solver is used to check which of its branches can be reached from the current state. It then adds all reachable branches as new states to the queue.
By checking which paths are possible to reach and then adding all reachable states to the queue the Symbolic execution algorithm aims to eventually reach any possible state.

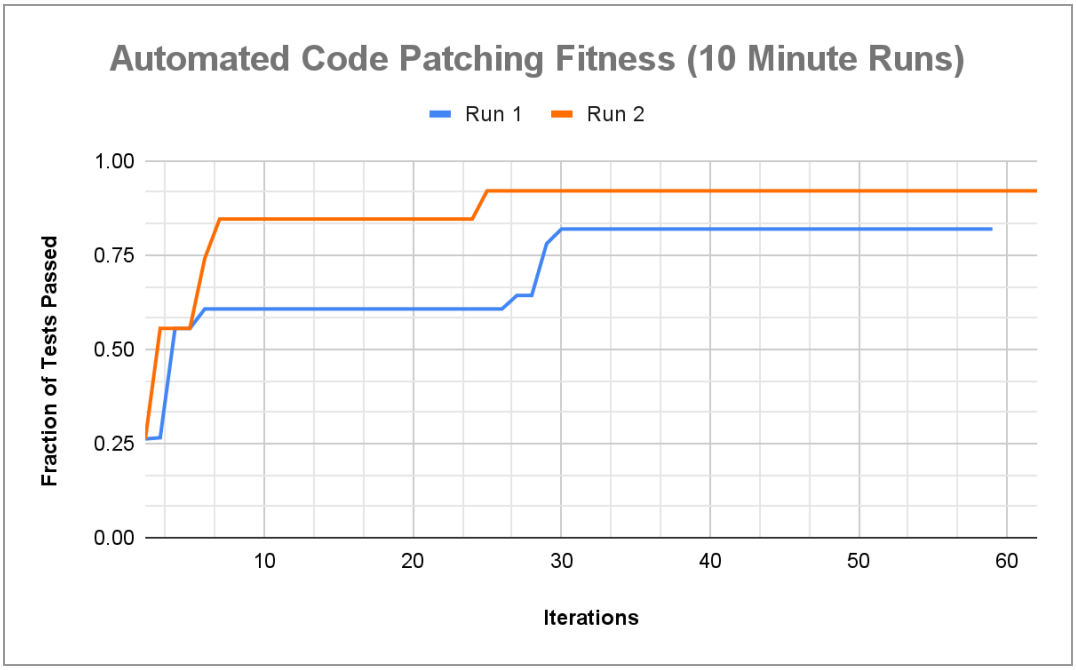
Problem	11	12	13	14	15	17
Num. Branches Covered	298	392	401	806	2078	4228
Num. Errors Triggered	17	12	22	14	35	30

Results Symbolic execution

One thing that matters is setting the initial trace length. Short traces are less interesting to check as they get covered by longer traces, but take too long a trace and executing a single trace takes too long meaning you try less traces in total and thus maybe find less bugs. We chose to start with a length of 1 and increase it whenever we run out of traces to check. This is a sort of middleground that starts off small but tests bigger traces if time permits. It might have been better to set the start slightly bigger to skip a bit of the initial uninteresting traces. Furthermore we keep track of which branches have already tested for a similar trace to avoid rechecking the same code.

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

The aim of the model learning algorithm is to learn a state machine for the given problem by only looking at input/output behavior. States for the state machine are defined as follows: If two or more traces (list inputs) lead to the same state then whatever input comes after those traces must give the same output.
The key idea of the algorithm is to keep track of the access string, which is a trace with minimal length that reaches a certain state. It also keeps track of all one symbol extensions, which are access traces with any valid operation added, and it tracks special traces that are used to differentiate states.
It then performs the following steps.

1. Check for completeness and consistency. That is every state should have exactly one outgoing transition for every valid operation.

2. Check for correctness. This uses an equivalence checker like the W method which basically runs a bunch of random traces based on the access strings and tests whether the model and the program output the same values (have the same behavior)

Problem	1	2	4	7	Pin
States	35	55	88	107	6
Membership Queries	781832	2201891	3686717	21430242	1332433
Runtime (ms)	53255	15307	253484	1478551	96634

Results Our L* Implementation

The decision that has the biggest impact on the performance is the way we implemented the counterexample processing. It adds all prefixes of the counterexample to the list of access traces (S) (as one of them must get to a new state) and then uses the consistency check to fix anything that might be going wrong from having traces there that might lead to the same state. This is an easy way to implement this but it has the disadvantage of greatly increasing S which in turn forces many more queries to be run to fill out the table.

2 AFL

2.1 Comparison: AFL, Smart Fuzzer, Random Fuzzer

Problem	11	12	13	14	15	17
Num. Paths found	76	89	117	137	139	93
Num. Errors Triggered	7	0	6	8	2	7

Results AFL

We ran AFL on the same RERS problems as our own code and we ran both programs for five minutes total. For AFL this time was tracked by the displayed runtime and the program was halted after it had passed five minutes.
AFL from our testing finds a significantly lower number of unique errors than either our random or smart fuzzer. It is quite likely that not running AFL in persistent mode (as is default in the repository) can have an enormous impact on the speed of AFL.

The reported errors found here is the number of unique errors found by the python script provided to check the AFL output. This is lower than the number of unique crashes reported by AFL so some of these unique crashes must be from the same error.
As for the number of unique branches visited it is hard to say for sure as AFL does not keep track of branches visited but rather keeps track of paths. We can however say that if you reach more error codes then you likely also reach more branches as the errors are all on different branches. Thus the results show that our implementations are able to reach more error codes and thus more unique branches.

2.2 Comparing Traces

One thing is that AFL generates quite short traces when compared to our generated traces. Furthermore it also can include characters in the traces that are not part of the input characters. We do provide the input characters to AFL but it might sometimes use other characters and those can then be found within traces it runs.

3 KLEE vs AFL

3.1 How They Work

AFL is a brute force fuzzer which is guided by a genetic algorithm. It loads the initial provided test and then repeatedly:

1. Take the next input and attempt to trim it such that the measured behavior is still the same.
2. Mutate the input.
3. If any mutated input leads to a new state then add them to the input queue.

It then implements many smart optimizations such as a corpus of interesting test cases that can then be used for seeding or other resource intensive testing regimes. Because AFL is random it allows it to find branches that are hidden deep in the code which techniques that try to explore every path might take a long time to reach.

KLEE is a symbolic execution engine. The core of KLEE is an interpreter loop that selects a state and then symbolically executes a single instruction in that context. This loop then continues until there are no states left or some timeout is reached. If KLEE encounters a branch it queries a constraint solver to see if the branch is always true or false. If it can take both values then two states are created, one where the branch is true and the other where the branch is false. It does the same for any ‘dangerous statements’ statements that could produce an error like division. It then also tries to find out if the error can be triggered. If so it creates a test case for it. It also creates a branch where the error is not triggered to explore the rest of the program. What is good about

symbolic execution is that because it verifies whether a branch is reachable it does not get stuck on hard to or impossible to reach branches

3.2 Results

Problem	11	12	13	14	15	17
Num. Errors Triggered	18	13	22	15	39	7

Results KLEE

Note for these results is that KLEE does in usual runs give a memory error. We tried to fix it by giving it no memory cap but it would still run into this issue. We expect KLEE to be able to find more errors if it was able to complete the 5 minute runtime but at the moment the most it could usually get to was ~2:30.

3.3 Analysis

KLEE with the recommended setup KlEE symbolically checks all inputs up until input length of 20. Same as with AFL KLEE was run for a maximum of 5 minutes on each problem.
What is very notable about KLEE is that it finds nearly all its errors within the first minute of operating with the latest time it found a new error being after ~85 seconds for problem 15. AFL was still finding new errors much later in its runtime for example 270 seconds in for problem 17. KLEE also found its first error more quickly on average.

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.

Results ASTOR

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

[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.

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 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

In the table below are the results from running the LearnLib implementation of the TTT algorithm:

Problem	1	2	4	7
States	35	55	88	-
Membership Queries	4687923	13822399	23210263	N/A
Runtime (ms)	326990	987913	1667203	N/A (>2h)

Results LearnLib TTT

Our method is able to find the same models as the TTT algorithm. You can see this because the number of states in both models is the same. We also know that for both models some equivalence checker was used to check if the models were equivalent to the program under test and these two statements combined give us good grounds to believe that both models are equivalent.

We can see that the TTT implementation uses significantly more membership queries when compared to our own method. We believe it might be the case that the TTT method does a more rigorous equivalence check and thus uses many more queries to find the model. We think this might be because when we would increase the W parameter then our method would also take much more queries and much longer to complete.