AI for Software Testing and Reverse Engineering (CSE4110) - Group 15 -

Andrei Ionescu & Kevin Nanhekhan

April 28, 2023

Artificial Intelligence can be used in automated software testing and reverse engineering. This, for example, is through the use of Machine Learning to uncover a model from data that explains program behaviour as to find errors that are otherwise difficult to find. The programs from the *Rigorous Examinations of Reactive Systems* (RERS) 2020 challenge were used to test the various techniques and algorithms described in this report. The 'Readme' inside the included code submission contains the necessary steps followed to run the code and tools used for all the sections of this report.

1 Algorithms

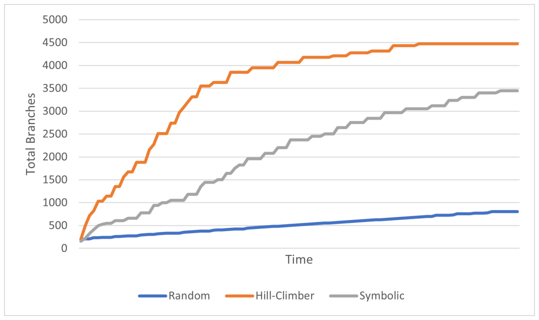
1.1 Hill-Climbing

A Fuzzing (or Fuzz testing) forms an automated testing technique that utilises random, invalid or unexpected input to probe programs with the hopes of detecting problems such as defects and vulnerabilities. The Random Fuzzer is a basic form of fuzzing that generates random inputs in an attempt to brute force the programs to cause problems. To improve its efficiency, a Hill-climbing algorithm can be used that selects the optimal input traces that have a lower computed sum of branches compared to other found traces while still triggering as many possible error codes. The selected trace will be mutated and used in the following iteration to see if an even lower sum of branches can be achieved. If no more optimal trace can be found, a random trace is generated to avoid getting stuck in a local optimum.

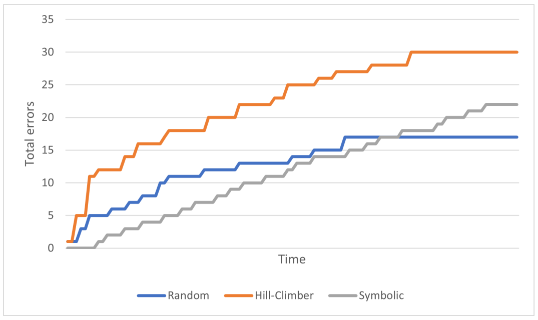
Reachability Problem	Random		Hill-Climber	
	#Error codes	#unique branches	#Error codes	#unique branches
11	18	306	18	306
12	4	348	6	356
13	15	225	22	403
14	3	153	14	604
15	1	551	39	2116
17	17	806	30	4471

Table 1: Results for Random and Hill-Climber Fuzzers on each of the RERS problems.

For comparison, the Fuzzers were ran for 5 minutes each on the RERS problems 11-15 and 17 with a set trace length of 20. The Hill-Climber only outperformed the Random Fuzzer by a slight margin which should not be the case, as the Hill-Climber follows a more guided path using mutations of the optimal trace instead of using random traces. This led to some adjustments having been made. The first concerns that a single mutation does not lower the branch-distance causing the Hill-Climber to act similarly to the Random Fuzzer. Therefore multiple mutations are now generated and looped through before using a randomly generated trace. The second adjustment concerns that of better trace mutation through the use of additional alternatives besides replacing operators, namely addition and deletion as new mutation operators. Rerunning the problems, see the results in Table 1 and Figure 1, the Hill-Climber outperforms the Random Fuzzer with a marginal difference. The reason for this is that the Hill-Climber is able to cover the branches more effectively in the same amount of time by selecting the optimal trace more often and therefore encountering more error codes.



(a) Number of unique branches encountered over time.



(b) Number of errors encountered over time.

Figure 1: Convergence graph for both Fuzzers and Symbolic Execution on problem 17.

1.2 Satisfiability Solver

Symbolic Execution forms an alternative to Random and Hill-climbing Fuzzer and is used to generate high-coverage test suites. It does this by monitoring program branches and generating path constraints that are forwarded to a Satisfiability modulo theories (SMT) solver, such as Z3 SMT Solver. The Solver checks the satisfiability of these constraints and generates concrete input values for possible program input traces to explore many program paths. Whenever a new branch is encountered, the line number and branch value are stored to indicate it has been visited and also a satisfiability check is run on the flipped branch condition. In the case it is satisfiable, an extra random symbol is added to the inputs to perform more of a breadth-first search and afterwards stored in a PriorityQueue. This Queue sorts them from shortest to longest and uses these traces for following iterations unless the queue is empty, then it will use a randomly generated trace. In the case it is unsatisfiable, the trace is stored in a HashSet to prevent it from being rerun. Finally, a constraint is added to the branches to follow the same program path for any subsequent branches that use that constraint.

Reachability Problem	Symbolic	
	#Error codes	#unique branches
11	18	306
12	10	369
13	18	384
14	15	604
15	13	1106
17	22	3443

Table 2: Results for Symbolic Execution on each of the RERS problems.

The experiment utilized the same setup as the Fuzzers. When comparing the results presented in Table 1 and Table 2, it shows that Symbolic Execution outperformed the Random Fuzzer but was not as effective as the Hill-Climber. To gain a deeper understanding, convergence graphs were generated for one of the problems (problem 17), as shown in Figure 1. The Fuzzers exhibit a logarithmic growth due to their nature as black-box testing techniques that can quickly generate and mutate test cases to identify simple input sequences. Making them highly effective for finding easy-to-trigger errors in a short time frame. In contrast, Symbolic Execution exhibits a more constant growth pattern due to its nature as a white-box testing approach, which requires an in-depth understanding of the program's internals and analyzes all possible program paths. Consequently, it involves a more complex and time-consuming process, leading to worse performance in the 5-minute testing period. However, if given more time, it is likely to produce better results, as indicated in the graphs by the growth in the number of branches and errors covered when compared to the Fuzzers. This is because Symbolic Execution is capable of exploring all program paths, making it better suited for finding more complex input sequences that trigger certain errors. It should be noted that Symbolic Execution may not generate inputs for all program paths if the SMT solver cannot efficiently solve the constraints along a feasible path. Therefore, to maximize the effectiveness of testing, it is important to understand the limitations and complement it with other testing techniques, such as that of the Fuzzers.

1.3 Genetic Programming

Evolutionary algorithms can be used to automatically locate and patch faults in software by leveraging the guided random process to construct new inputs or tests that trigger new code branches. The process involves using a fault localization strategy that employs the Tarantula score to compute the frequency of operators used in failing tests, ranking identified faulty operators based on their score, generating potential patches, and using a genetic algorithm to select the best patch through crossover and tournament selection. To elaborate on the last, mutation and crossover are used to generate new individuals by modifying existing ones to maintain diversity in the population and avoid local optima. Mutation does this by performing random changes to the operators and crossover swaps two or more operators at randomly chosen positions from the tournament winners (the ones with the best fitness value) to create a new one. This creates a new population that can be used in the subsequent iterations. The success of the algorithm depends on the fitness function, mutation and crossover rate, and population size, with high rates leading to premature or poor convergence, and low rates resulting in a lack of diversity. Therefore varying population sizes are used to address this issue. The initialization has been done with a lower population size as only copies of the operators are used and later on the population size is increased as the operators are at that point mutated and no longer the same copies.

Reachability problem	Best Fitness (max = 1.0)	Percentage Patched
1	1.0	100%
4	1.0	100%
7	1.0	100%
11	0.85	10%
12	0.49	23%
15	0.63	38%

Table 3: Comparison problem performance and percentage of faults patched.

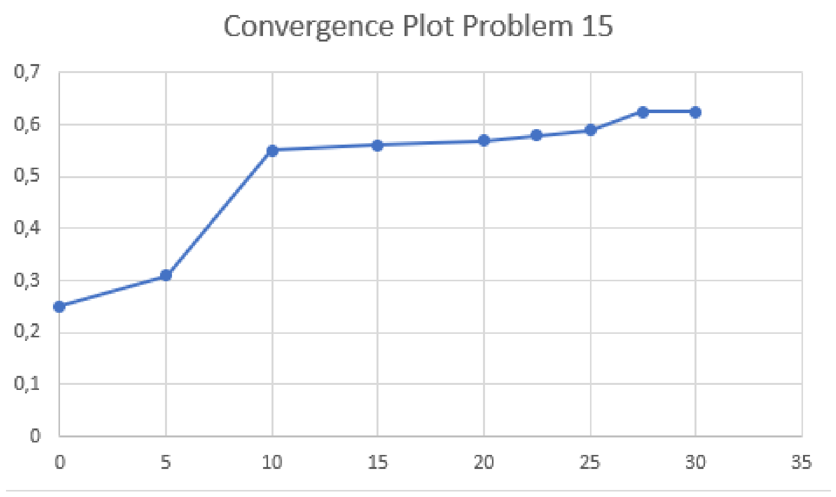


Figure 2: Convergence Graph for Problem 15

1.4 Active Learning

Active State Machine Learning involves modeling the behavior of a software system through observation tables and finite-state machines, such as Mealy machines. To ensure accuracy, it is crucial that the observation table containing information on the input-output relationship is both consistent and closed. Inconsistent tables violate the assumption of determinism, while non-closed tables fail to achieve completeness, both of which affect the accuracy of Mealy machine-generated results for certain inputs. Once a closed and consistent model has been established, the L* algorithm can be used to further update the model by iteratively ensuring that any input/output is equivalent with the observed data. This involves performing an equivalence check that verifies newly generated hypotheses to possibly find a counterexample. While a simple Random Walk Equivalence Checker can be used for this purpose, the W-method is a more sophisticated approach that generates all possible combinations of words consisting of an access trace, input sequence of length w, and a distinguishing trace. When a counterexample is found, the next step is to process it and update the observation table. This update includes the output of the system for a specific input, ensuring that the table reflects the system's behavior and is consistent with the counterexample. This process continues until no counterexample can be found, which indicates that the generated hypothesis model is equivalent to the observed data but it does not guarantee it is of the correct system model.

Learning the models through the RandomWalk and W-method from the RERS challenges 1, 2, 4, 7 and the ProblemPin, we get the following results:

Reachability Problem	Random		W-Method	
	#States	#Membership queries	#States	#Membership queries
1	27	31393	27	227558
2	37	50440	25	140418
4	63	67477	34	219770
7	57	81720	32	918271
ProblemPin	1	10052	1	111210

Table 4: Results for Symbolic Execution on each of the RERS problems.

The pin problem should look like this:

- q0 entry state
- When the first digit is entered, the machine transitions from q1 to q2 (invalid)
- When the second digit is entered, the machine transitions from q2 to q3 (invalid)
- When the third digit is entered, the machine transitions from q3 to q4 (invalid)
- When the fourth digit is entered, the machine transitions from q3 to q4 (valid)

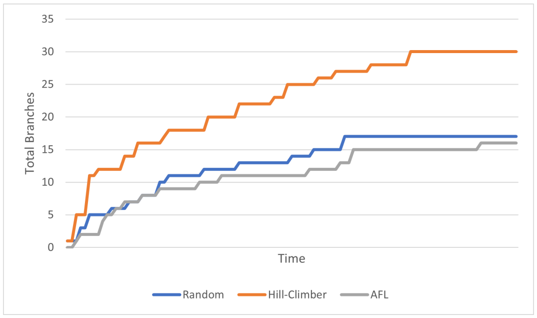
2 AFL

AFL [Zal] is a state-of-the-art fuzzer that is widely used by security researchers and developers. It uses genetic algorithms to generate input data for a program, then employs instrumentation to measure code coverage and guide the fuzzing process towards areas of the program that have not been thoroughly tested.

Reachability Problem	AFL	KLEE
	#Error codes	#Error codes
11	17	18
12	13	13
13	22	22
14	15	15
15	5	39
17	16	30

Table 5: Results for AFL and KLEE on each of the RERS problems.

The results obtained by running AFL on the RERS problems (11-15 and 17) for 5 minutes each, similar to the Random and Hill-Climber Fuzzer, are presented in Table 5. A more detailed analysis focusing on Problem 17 is shown in Figure 3. The results indicate that AFL was not able to reach more errors and thus most likely also unique branches on the same RERS problems within the same time constraints as the other Fuzzers. This can be explained by the fact that AFL uses a more sophisticated and resource-intensive approach based on coverage and mutation strategies to generate test cases, which may require more time and resources compared to the simpler strategy of randomly generating and modifying inputs used by the other Fuzzers. As a result, the other Fuzzers achieved better results in the short 5-minute testing period. However, given the aforementioned reasons, it is suspected that AFL may achieve more optimal results given more time. This idea can be supported by looking at the graph of errors encountered (see Figure 3a). The graph suggests that AFL is still in its initial phase of its logarithmic growth, with a fast growth rate over time. In contrast, the other Fuzzers show a slower growth rate, suggesting they have reached their maximum potential within the given time constraints.



(a) Errors encountered over time.

Error codes	Hill-Climber	AFL
0	O H O E N A Q O B	H H O E
2	J D B P J N R P	B P G
11	H L C G K J G O P	H D M C O A M C A
19	B H G A T F L N Q A	B H G H A
23	G J Q B E R L F Q	B E O
46	B N G A S M S N P T	B G R G A G P G
53	O M H P A F J M	H H J B
61	N B G L I C P J B	B G G C
66	R F E C C B G P	F B
69	T N K F I D P M	T I

(b) Input traces per triggered error.

Figure 3: Results for Random Fuzzer, Hill-Climber and AFL on problem 17.

In Figure 3b, a subset of the triggered errors have been displayed for the (shortest) input traces by both the Hill-Climber and AFL. It should be noted that Hill-Climber was rerun with tracelength of 10 for easier comparison. In general one can see that although the input traces generated by both fuzzers were not identical, they contained similar input characters in the same order. For example in triggering *'error code 69'*, both Fuzzers both contain the inputs *'T'* and *'I'* which also appear in that order. Furthermore, while not fully shown in the table, AFL was also better in its mutation as it was able to generate traces with a more varied lengths (around 2 to 14 characters) compared to Hill-Climber which had few characters more or less than its predefined tracelength. This can further support the idea that AFL, given more testing time, will have more optimal results in encountering errors as it is better in generating more diversified input traces which is a desirable trait in fuzzing.

3 KLEE

In the previous section, AFL was discussed, a fuzzer used to generate random inputs for a program to find unexpected behaviour. While AFL is effective at finding crashes and simple bugs, it may miss more complex vulnerabilities that require specific input patterns. This is where Symbolic Execution, such as KLEE, can be particularly effective. Unlike Fuzzers, Symbolic Execution explores all possible paths of a program, rather than a subset of program paths. It uses symbolic values to represent all possible values that a variable can take and generates constraints on these symbolic values to find concrete values that satisfy those constraints. This allows for test case generation that covers all possible paths of a program, even those that may be difficult or impossible to reach.

Compared to other Symbolic Execution tools, KLEE [CDE08] is a state-of-the-art symbolic execution engine that is based on the LLVM infrastructure. LLVM is a popular compiler infrastructure that provides many powerful analysis and optimization features. KLEE leverages LLVM to perform sophisticated analyses and optimizations that other Symbolic Execution tools may not be able to. Additionally, KLEE is particularly useful because it supports the use of multiple constraint solvers (such as Z3 and STP), which is helpful if one solver is unable to solve a particular set of constraints. Lastly, KLEE uses a 'path-based' approach that explores different execution program paths while keeping track of the symbolic values of variables. This can be more efficient than other approaches (e.g. 'state-based' approach) which explores all possible combinations of program states. All these features make Klee a powerful tool for identifying complex vulnerabilities in software.

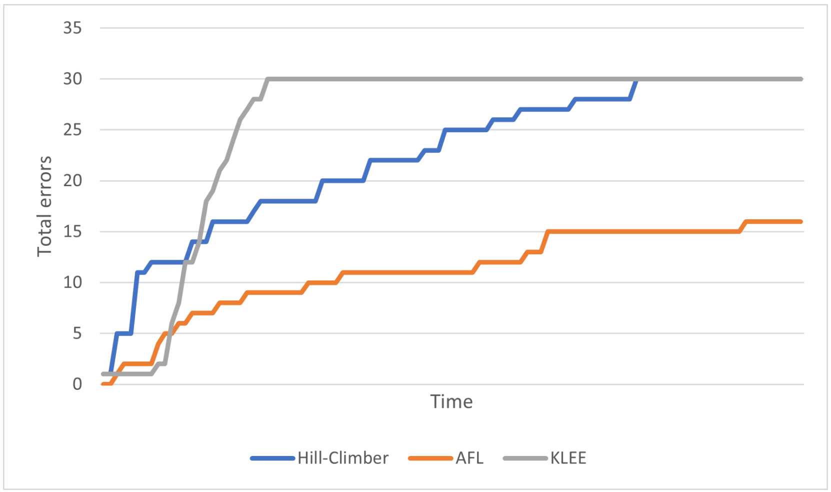


Figure 4: Convergence Graph for AFL and KLEE in the number of errors encountered on problem 17.

The results of running the same set of RERS problems for 5 minutes each have been documented in Table 5. For problem 17, a more detailed overview of the errors encountered over time has been provided in Figure 4. It is evident from the table and graph that KLEE performs significantly better than AFL, managing to detect most, if not all, known errors in the RERS problems like the Hill-Climber but within a shorter time frame. There is a stark contrast in performance between these results and the results of Fuzzers and Symbolic Execution discussed in section 1. This could be attributed to a difference in efficiency in the implementations and configuration. It is highly plausible that our implementations are not as sophisticated as state-of-the-art tools, making the process of triggering errors more time-consuming. It is important to note that KLEE's initial phase takes some time to trigger errors, as seen in Figure 4 where AFL finds more triggers at the start. However, due to its efficient implementation and configuration, KLEE as a white-box testing approach can explore all program paths effectively, leading to the quick detection of a significant number of errors. This is in contrast to AFL which in the end is a Fuzzer and black-box testing approach only generates input based on the feedback of a target program's execution. In the end, the performance of these approaches can vary significantly depending on the program being analyzed as well as the configuration of the tools themselves.

4 ASTOR

Astor [MM16] is an automated software repair framework, used in research studies and competitions, that automatically generates and validates patches for software bugs without requiring manual intervention. It uses program analysis, search algorithms, and patch validation techniques to automatically generate patches for bugs.

ASTOR was run on the compiled Maven project for each (buggy) RERS problem using command line parameters (as specified by the provided documentation in the lab) that specify the patch generation engine, the location of the source and test code, the maximum execution time, the maximum number of generations, the repair scope, the population size, and the mutation operators to use. The output of ASTOR was located in the output ASTOR folder, which contained a log file with the results of the search process, a list of the generated patches, and other metadata about the search.

The generated patches were in the form of modified Java source code files that attempted to fix the bugs in the original program. These patches could be manually inspected and tested to verify their effectiveness.

ASTOR also provided information about the search process, such as the number of candidate patches generated, the fitness of the best patch, and the time taken to execute the search. Overall, the analysis with ASTOR aimed to demonstrate the potential of automated program repair techniques to improve software reliability and reduce the time and effort required for manual debugging.

Below, in the table 6, you can see how many patches the ASTOR found and in how much time. Some problems did not finish executing after executing ASTOR on them for a few hours (even though the budget parameter was set to 5 minutes).

Problem No.	Time (min)	Suspicious lines No.	Patches
1	5	16	13
2	240	N/A	N/A
5	300	N/A	N/A
11	5	17	26
12	55	122	4

Table 6: Problems ran on ASTOR - time budget 5 minutes.

4.1 Patch Analysis

In this section, we set out to investigate the effectiveness of the ASTOR automated patch generation tool on the RERS benchmark suite. Previously, we first ran the RERS benchmark problems through ASTOR, generating a set of candidate patches for each problem.

We will manually analyze the generated patches using a diff-tool to compare the original RERS problems, their buggy versions, and the patched versions. We will answer whether ASTOR generated meaningful patches and evaluate the correctness of the patches.

Our analysis showed that ASTOR generated some meaningful patches, but many of the generated patches were incorrect and failed to fix the underlying problems. We then investigated the reasons behind the success and failure of the generated patches, considering factors such as the complexity of the problems, the nature of the bugs, and the quality of the input test cases as seen below, by analyzing Problem 1 with an actual example from the code.

4.1.1 Problem 1

Original (line 30):

```
if((((a691849188 == 4) && ((input.equals("iB")) && cf)) && (a547336540.equals("e"))))
```

Original Buggy problem (line 30):

```
if (((a691849188 != 4) && (input.equals("iB") && cf)) && a547336540.equals("e"))
```

Patched Buggy Problem (line 30):

```
if (((a691849188 != 4) || (input.equals("iB") && cf)) && a547336540.equals("e"))
```

As seen above, the patched version switches the second operator in the expression (the AND) to an OR. The buggy version also contains another change, in the first operator (equal turned to not equal). This type of patch is not necessarily considered a useful patch, but we cannot draw any conclusion without a deep understanding of the codebase. This change can treat the symptoms of a bug, or a sequence of bugs that occur in the code. By looking into more details about the patches of problems 1, 11 and 12 we reached the following conclusions in order to answer the assignment's questions:

- The code is highly obfuscated, it can be challenging to understand how the patches have changed the original code. This can make it difficult to assess whether the patches have resolved the issue or introduced new problems.
- It's extremely important to consider the context in which the bug occurred in the original code. Without this context, it can be challenging to determine whether the patches have actually fixed the underlying problem or simply addressed a symptom of the issue.
- Since RERS focuses on reachability problems, it is hard do conclude what is the expected behaviour when more than one thing changes into a problem (e.g. random bugs).
- Since the RERS problems are somehow complex when a bug is patched, it can be challenging to determine whether the change has unintended consequences elsewhere in the codebase.

- The introduction of random bugs (like the ones in RERS buggy problems) can make it even more difficult to determine whether the patched code is meaningful. In some cases, a program with a random bug may still produce the correct output, making it difficult to distinguish between the original code and the patched code. This can lead to a false sense of security that the patch has successfully fixed the issue, when in fact it may not have.

4.2 Conclusion

To sum up, auto-patching bugs can be a useful tool for addressing software vulnerabilities, but it's important to carefully evaluate the effectiveness of the patches to ensure they are truly meaningful. This requires a deep understanding of the codebase, in our case the RERS problems, the context in which the bug occurred, and the potential impact of introducing new code.

5 LearnLib

LearnLib [RS06] is a state-of-the-art tool used in research and education for learning state machine models using active learning, which allows for faster learning by interacting with the system. It also includes tools for visualizing learned models, analyzing their properties, and working with input/output sequences. LearnLib has a more efficient method for learning models than L* algorithm, namely the Table-driven Top-down Tree (TTT) algorithm [Lea].

5.1 L* vs TTT

Both L* and TTT are designed to learn deterministic finite automata (DFA) and Mealy machines. Yet, they use different strategies for building and refining the hypothesis. L* uses a table-filling algorithm that constructs a complete table of all possible input sequences and their corresponding outputs. TTT, on the other hand, constructs a tree-based model that partitions the input sequences into equivalence classes based on the observed output.

In terms of performance, TTT is generally faster and requires fewer queries in some cases (see table 7) than L*. However, this efficiency comes at the cost of a more restricted hypothesis space. TTT can only learn DFA, while L* can learn Mealy machines, which are more expressive and can capture a wider range of system behaviours. Moreover, L* is more robust to noise and errors in the input-output mapping.

Therefore, it is possible that when is L* used, we may find models that TTT cannot learn due to the differences in the hypothesis space and the search strategies. For the purpose of comparison, we used a Python library called a network that is able to see if the graph generated by our approach is isomorphic to the one created by LearnLIB using TTT. Since we set up a time-out for our approaches it is a bit hard to compare them just on the graph they generated. During the tests we performed, we observed that LearnLIB take a very long time to search for a counterexample - e.g. for problem 2 it took more than 7 hours to finish executing. On the other hand, the learning phase is quite fast. Since the problems grow in complexity (e.g. problem 1 to problem 5) we decided to use a smaller value for the w parameter.

5.2 Membership queries

In general, the TTT algorithm in LearnLib is known to require fewer membership queries than the L* algorithm. This is because TTT uses an equivalence class partitioning strategy that reduces the number of queries by eliminating redundant queries [Lea].

Therefore, the number of membership queries required by our implementation of L* may depend on the complexity of the target model (aka. RERS problem), the size of the input (which is somehow fixed for the task we perform) and the output alphabet, and the specific selection strategy and stopping criteria used. It is possible that L* may require more queries than TTT for learning DFA, but it may require fewer queries for learning Mealy machines. The exact comparison would depend on the specific experimental setup (the parameters we used) and the characteristics of the problems we applied it to. Below, in the table 7 you can see a comparison in terms of membership queries for different RERS problems.

Problem No.	Membership # Queries L*	Membership # Queries TTT	w parameter
1	921974	4687923	4
2	2508232	13822399	4
4	2134665	23210263	3
Pin	11210	11120	3

Table 7: Membership Query No. comparison

Problem 7 didn't terminate using LearnLib after more than 6 hours, so we decided not to include it in the table above.

References

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[Lea] LearnLib. The ttt algorithm: A redundancy-free approach to active automata learning — learnlib.

[MM16] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA*, 2016.

[RS06] Harald Raffelt and Bernhard Steffen. Learnlib: A library for automata learning and experimentation. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, pages 377–380, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Zal] Michal Zalewski. Google/afl: American fuzzy lop - a security-oriented fuzzer.