Proceedings of Machine Learning Research 217:142–156, 2023 ICGI regular paper

Detecting Changes in Loop Behavior for Active Learning

Bram Verboom B.O.VERBOOM@STUDENT.TUDELFT.NL
Simon Dieck S.DIECK@TUDELFT.NL
Sicco Verwer S.E.VERWER@TUDELFT.NL
Delft University of Technology

Editors: François Coste, Faissal Ouardi and Guillaume Rabusseau

Abstract

Active automaton learning is a popular approach for building models of software systems. The approach forms a hypothesis model from observations and then performs a heuristic equivalence query to check if the learned model is equal to the model under test. The current methods for equivalence queries, however often fail to find counterexamples when encountering loops, one of the most common control structures in software. We introduce two novel equivalence checkers that better handle loops. One extends the well-known W-Method, and the other uses symbolic execution. Both methods are tested on RERS challenge problems. We show that our approaches find more counterexamples on suitable problems and thus learn more accurate models. We further test our symbolic execution approach outside active learning and show that it finds more errors than the state-of-the-art method Klee on several problems.

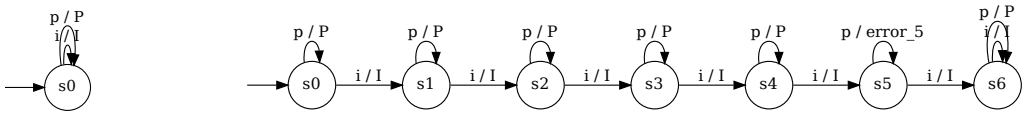
Keywords: Active Learning, Equivalence Checker, State Machines, Mealy Machines, Symbolic Execution, Symbolic Loop Execution, Loop Detection, W-Method

1. Introduction

Active automaton learning has become a key technique for finding behavioral bugs in software systems. Analyzing models that were learned from software has successfully been used to detect bugs in real implementations of for instance TLS protocols by (De Ruiter and Poll, 2015) and (Fiterau-Brostean et al., 2020), SSL (Sivakorn et al., 2017), and OpenVPN (Daniel et al., 2018). The approach consists of two main sub-problems: Building a hypothesis model from observations and checking if the hypothesis model is equivalent to the model under test. Our main focus is on the equivalence queries. Existing equivalence checkers such as the W-method (Chow, 1978) focus on equivalence checking of a model by testing every transition up to a certain depth w. This approach struggles, however when encountering loops, which often only show change in their behavior after some possibly high number of iterations. To illustrate this problem and motivate our work we will use a simple running example found in Figure 1. The program takes in characters i and p and counts how many i's were input so far. When reading a p it checks if the number of i's it has seen exceeds some given limit and if it does outputs an error. Of course, this example is highly simplified, but the structure of a loop that changes its behavior after some amount of iterations is very common. The most common example is a loop that simply terminates after a certain amount of iterations. If we set the limit in our example problem to 5 but choose a w, which is smaller than 5, for the W-method an active learner will learn the

```
int main(int argc, char** args) {
  int limit = atoi(args[1]); // Target to reach
  int i = 0; // Internal state
  int j = 0; // Loop variable
  char symbol; // Character in input
  char* trace = args[2]; // Input: array of symbols
  while ((symbol = trace[j++]) != 0) { // Get next symbol
    if (symbol == 'i') { i += 1; }
    else if (symbol == 'p') {
      if (i >= limit){ assert(0); }
    } else { assert(0); }
  }
}
```

Figure 1: C Program used for comparing Klee (JDart uses similar Java code) and SymLoop.



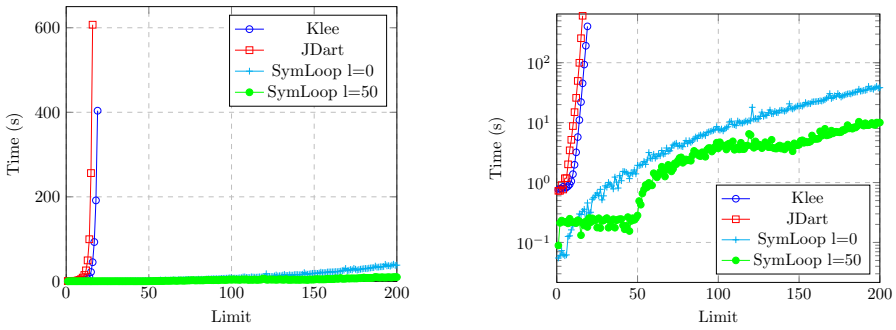
(a) Change not found

(b) Change in loop behavior found

Figure 2: Learned models of the same system. The learning process that produced the left model was unable to find a change in behavior after 5 iterations through the 'i' loop. The learning process resulting in the right model does capture this behavior.

model seen in Figure 2(a), while the correct model can be seen in Figure 2(b). To address this problem of the W-method, we introduce a straightforward adaptation to better handle looping structures which we call *Loop-W*.

Often we also have access to the source code of a system under test. Symbolic execution is a method that utilizes a software's source code to discover new paths through the software. It uses "symbolic" variables and satisfiability modulo theories (SMT) solvers to discover behavior that has not been covered yet. We can also utilize symbolic execution to guide equivalence queries. The state-of-the-art symbolic execution tools Klee (Cadar et al., 2008) and JDart (Luckow et al., 2016) however, suffer from a so-called path explosion problem (Baldoni et al., 2018), that leads to significant running times on problems with looping structure. In our example, this is due to a symbolic execution creating for each iteration inputs that meet the "$symbol ==' i'$"-branch as well as the "$symbol ==' p'$"-branch. Thus, it generates all possible input traces. Though for a human it is clear that e.g. the sequences "*ip*" and "*ippp*" result in the same behavior and wouldn't need to be tested separately.



(*a*) Normal Scale (*b*) Log Scale

Figure 3: Time needed for running symbolic execution on the program from Figure 1 to find an error for a certain limit. For Klee we stopped running past a limit of 19. JDart crashed past a limit of 16, and did not find the error.

We introduce *SymLoop* a symbolic execution approach that addresses these looping structures. It identifies loops and then tests how many iterations are necessary for behavioral change to appear, up to a limit l. Even setting $l = 0$ can save significant time, as it allows ignoring self-loops that never introduce behavioral change (the "*ip*" and "*ippp*" example given before). How much running time can be saved with this approach on our example problem is illustrated in Figure 3.

We implemented both the Loop-W and SymLoop methods in the LearnLib active learning library (Merten et al., 2011) and tested their performance on problems from the 2020 RERS challenge (Howar et al., 2021). There we show that our methods reach the same amount of code branches as the state-of-the-art methods on problems without these looping structures and discover more code branches on suitable problems than the state-of-the-art methods.

The main contributions of our paper are thus as follows:

- Loop-W, an extension of the W-method, that can handle loops.
- A method to use symbolic execution for equivalence queries in active learning.
- SymLoop, an extension to symbolic execution, that runs much faster on looping structures.
- Experiments that show the performance of Loop-W and Symloop.

2. Background

Here we will give a somewhat simplified introduction to the concepts that are used in this paper. For notation, we will use '$\cdot$' to denote string concatenation and λ for an empty word, s.t. $w \cdot \lambda = w$ for all words. Further, we use $|w|$ to denote the length and therefore the number of characters in a word w and similarly $|S|$ to denote the number of elements in a set S.

The model learned by the active learning methods of interest to this paper is a deterministic finite automaton (DFA). It is defined by the tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ a start state and $F \subseteq Q$ the set of accepting states. We use Σ^* to denote the set of all possible words over the input alphabet Σ and extend the transition function δ to $\delta : Q \times \Sigma^* \to Q$, by giving for a symbol $u \in \Sigma$ and a word $w \cdot u \in \Sigma^*$ the recursive definition $\delta(q, w \cdot u) = \delta(\delta(q, w), u)$, with $\delta(q, \lambda) = q$ covering the edge case.

As a bit of extra notation, we take for a word $w \in \Sigma^*$ w^i to mean w is concatenated to itself i times.

Angluin introduced the **minimally adequate teacher (MAT) framework** for learning a DFA from a system (Angluin, 1987). The core of the MAT framework is a teacher and a learner. The learner constructs a hypothesis model based on membership queries. When it constructs a hypothesis model, the teacher is invoked to perform an equivalence query. The teacher verifies whether the system under learning matches the hypothesis model. If the teacher finds an inconsistency, a counterexample is returned to the learner to form a new hypothesis model. This process continues until the teacher can not find a counterexample for the current hypothesis. The hypothesis is then output as the final model. In the same paper, Angluin introduced the L^* learning algorithm. Since the introduction of this algorithm, new methods have been proposed to reduce the runtime, the memory overhead or the number of membership queries to learn a model.

In MAT learning, to check if a hypothesis is correct, an equivalence checker is needed. The most common equivalence checker is the **W-method** by Chow. The W-method verifies a hypothesis model by taking the access sequence of each state q ($a \in \Sigma^*$ s.t. $\delta(q_0, a) = q$) which together form the set $A \subset \Sigma^*$, the set of all possible sequences of length up to w over the input alphabet, which we name $W \subset \Sigma^*$ and all distinguishing traces of the model, which is denoted by $D \subset \Sigma^*$. W is constructed, s.t. $\forall q_1, q_2 \in Q, q_1 \neq q_2 : \exists d \in D : \delta(q_1, d) \neq \delta(q_2, d)$. These three parts are concatenated, and each sequence $a \cdot z \cdot d$ for all possible combinations of $a \in A, z \in W, d \in D$, is checked for equivalence with the model. Checking a single trace is done by resetting the system under learning and running the trace, if the output of the system matches the output of running that same trace through the model, the model matches the system. If the outputs do not match, a counterexample is found and the trace is given back to the learner to refine the hypothesis.

Symbolic execution uses the source code of a program to discover as set of input traces, that reaches every line in a program. It does this by creating a "symbolic" variable each time some assignment happens. For example having two lines of code, one that assigns $a = 1$ and a following one that assigns $a = a + 1$, would result in two symbolic variables $a_0 = 1$ and $a_1 = a_0 + 1$. If these assignments use symbols from some input sequence $S = s_0, s_1, \ldots, s_n$ they form a set of equations where the elements of S can be treated as unknowns. Symbolic execution first executes a program with an arbitrary input S. Each time a branch is encountered it uses the equation system it build to call an SMT solver and check if there exists an input S that would reach the branch not reached by the current execution. If it does exist, this input is added to a queue for future runs. This process is repeated, until either all lines of code have been reached or the input queue is exhausted. In reality, it is often stopped due to a time limit.

3. Related Work

We consider relevant literature in both active learning and symbolic execution.

Klee (Cadar et al., 2008) is one of the more used symbolic execution engines. Cadar et al. noticed that most of the time in symbolic execution is spent on solving SMT queries. Klee is optimized to reduce the number of calls to the SMT solver, or, if unavoidable, make the SMT queries simpler. Similarly to Klee, **SymCC** (Poeplau and Francillon, 2020) also focuses on reducing the overhead of the solver. Yet the main contribution is to reduce the overhead of interpreting the code through a compiler pass that includes the symbolic execution directly in the program.

Although both Klee and SymCC yield significant speedups, the path explosion itself is not tackled. However, there is research into ways of reducing the path explosion problem, specifically for loops.

Loop Extended Symbolic Execution (LESE) Saxena et al. (2009) introduces symbolic variables for the number of times each loop is executed. These trip counts are linked to the patterns in a predefined input grammar. In their work, they look for a linear relationship between the trip counts. Although the method works for the example shown in the introduction, the use of an input grammar and only considering linear relations is limiting.

Efficient Testing of Different Loop Paths by Huster et al. introduced a methodology for analyzing multiple different paths through a loop (Huster et al., 2015). By leveraging static analysis, possible loop paths are extracted from the program. By analyzing the read and writes for each iteration, they can create combinations of these iterations that affect each other. The results show that this approach is able to cover code with loops more effectively.

Efficient Loop Navigation for Symbolic Execution Obdržálek and Trtík (2011) is another method that tackles the same problem. Our example shown in the introduction closely resembles their example. Their approach creates chains and constraints representing executing loops based on loop counters. When solving for new paths which might require iterations through the loop, the system checks whether incrementing any of the loop counters improves the current solution. This process allows them to reach branches that require more loop iterations.

For active learning **TTT** was introduced by Isberner et al. (2014) to reduce redundancy in the observation table of L^* by keeping track of a discrimination tree for storing the discriminators. This results in a significant reduction in memory and also reduces the number of membership queries necessary to construct hypotheses.

Recent work by Vaandrager et al. (2022) introduces a new learning algorithm called $L^\#$. Instead of keeping track of an additional data structure such as an observation table, it directly constructs and operators on a partial mealy machine that includes all observations. Instead of focusing on equivalence, their work uses apartness. When two states are apart, the states are distinct in the hypothesis model. Apartness denotes a conflict in semantics. Their results show that $L^\#$ is not strictly better than other methods such as TTT by needing a comparable number of membership queries. Their method outperforms state-of-the-art algorithms by requiring fewer symbols for learning.

While not specifically mentioned as a purpose, works like the one from Drewes et al. (2011) sidestep the problem of loops by learning tree structures instead.

As for improving the equivalence queries, using **Adaptive distinguishing sequences** (ADTs) (Hierons et al., 2008), the total number of queries can be reduced. Whereas non-adaptive methods use all distinguishing sequences of a model, adaptive sequences check the behavior under the assumption that the system under learning is in the expected state.

4. Method

During equivalence checking in active learning, finding counterexamples is crucial to be able to update the hypothesis of the learner. In this paper, we focus on checking the loops in a hypothesis model. We propose two methods to perform this check: asking many membership queries by combining unrolled loops with equivalence queries: *the Loop-W method*, and symbolically executing the loop (requiring no membership queries): *SymLoop*.

4.1. The Loop-W-method

First, we want to motivate, why the W-method struggles with loops. Assume we have a loop of length d (The loop consists of d states and d edges in the DFA) and to observe new behavior, this loop needs to be traversed l times. This directly implies that w needs to be larger than $d * n$. But since all possible traces of length up to w are generated with the W-method, this already constitutes $|\Sigma|^{d*l}$ traces. This still gets combined with the number of access sequences, which is equal to the number of states, and the number of distinguishing traces, which in the worst case is also equal to the number of states. In total the W-method will therefore test $O(|Q|^2 * |\Sigma|^{d*l})$ traces.

The problem here is that even though we are interested in only specific long sequences, namely the ones that repeat a loop, we still generate all possible sequences with that length. We address this with our Loop-W-Method, by first identifying the loops, and then adding traces that repeat them as a whole. Identification can easily be done with a depth-first search (DFS). For adding the loop multiple times, we introduce an extra parameter l, which determines how many times a loop will be repeated by the method. Say we identified a loop that starts at state q and has the body b, meaning $\delta(q, b) = q$, we can construct an extended access sequence. If a is an access sequence for q, then $a \cdot b^i$ is also an access sequence for q, for all possible i. We use this to construct extended test sequences $a \cdot b_a^i \cdot z \cdot d$, for all possible combinations of $a \in A, z \in W, d \in D$ and $i \in [1, \ldots, l]$, where we define $b_a = \lambda$ if the state reached by a is not part of a loop and the body of the loop otherwise. A, W, D are defined as in the standard W-method. If we now want to reach new behavior that shows up after a loop of length d has been traversed l times, we can choose w independent of d and l and only need $O(|Q|^2 * l * |\Sigma|^w)$ test traces to reach this behavior. Moving l out of the exponent and eliminating d, allows us to efficiently explore the behavior of loops even for small choices of w. The Loop-W-method tests this extended set of traces and otherwise works the same as the regular W-method.

4.2. SymLoop

Learning models for systems where the source code is available allows additional techniques since the system is no longer a black box. Instead of the naive approach of checking every

iteration of a loop, we can use dynamic analysis techniques like symbolic execution. This allows detecting behavioral changes of systems after a number of iterations through a loop.

4.2.1. Execution Model

For the symbolic equivalence checker, we assume the execution model of the programs under learning. The execution model follows the input-output pattern of Mealy Machines. The program must contain one core loop that retrieves the input and produces an output. The program repeats this loop. An invalid input or reaching the end of the input will cause the program to terminate. In each input-output cycle, the internal state of the program may change. All operations are deterministic, so running the program again with the same input will yield the same output. The inputs are symbols from a fixed alphabet.

We based the execution model on the RERS Challenges (Jasper et al., 2019; Howar et al., 2021). The challenges are built to encourage combining research fields for better software verification. The problems are generated to be realistic problems of scalable complexity.

Due to our execution model assumptions, we can form path constraints for a single iteration through the input-output loop. These loop constraints represent following a specific loop path through the program.

This execution model simplifies the identification of loops but shouldn't affect the other results presented in this paper.

4.3. Symbolic execution for equivalence queries

For loops in hypothesis models, there are two possibilities. The first possibility is that the hypothesis is correct and the loop is a true self-loop and as such can be executed an arbitrary amount of times. The second possibility is that the loop will change its behavior after some amount of iterations and as such would need to be unrolled in the hypotheses model. Even using the Loop-W-method testing these possibilities using test traces is expensive and we can never be certain that it is a true self-loop or just no long enough trace was checked. Here symbolic execution can help. Given a state q, that is start and endpoint of a loop, we can use the access sequence of q, a, as an input for symbolic execution to identify the point where a loop starts. To find the point where the loop finished one iteration we can form the sequence $a \cdot b_a$, where b_a is the body of the loop in the hypothesis model. Using this information we can use symbolic execution to better reason about the internal workings of the program during the loop.

4.3.1. Self loops

When a loop path contains no assignments, the internal state of the system does not change. Repeating the same symbol will yield the same output. Loop paths with assignments that keep the variables at the end of the loop in the same state as before inputting the symbol have no effect as well. In both cases, the symbol that excites this behavior can be confirmed as self-loops in a learned model. The same process can be done for loops over multiple symbols. The loop constraint then captures a path over multiple iterations through the input-output loop in the program. Recall that symbolic execution for each variable in a program a creates a new instance of this variable each time it is assigned a new value. We will denote with a_i^j the content of the j-th variable after i assignments, where $i = 0$ will

be the initial assignment. Given the start point of a loop and the end of its first iteration, we will use s_j to be how often variable j has been assigned a value before the start of the loop, and t_j, how often it has been assigned a value after the loop. Accordingly $a^j_{s_j}$ is the value of variable j at the start of the loop and $a^j_{t_j}$ its value after one iteration of the loop. If $\forall j : a^j_{s_j} = a^j_{t_j}$ the loop must be a self-loop. We call this the self-loop constraint. Note that this is trivially true if $s_j = t_j$ but still holds if some assignments happened, as long as the internal state of the program is the same before the loop as after the loop. All assignments and conditions, that were done to reach the start of the loop with the access sequence can be used to form a constraint, which we call the path constraint. The same can be done for all assignments and conditions that are encountered during the loop, we call this the loop constraint. If the SMT solver used on the conjunction of the path, the loop and the self-loop constraint yields a satisfying assignment we can definitively confirm that the loop is a self-loop. The next section handles the case where no satisfying assignment is found.

4.3.2. Repeatable loop paths

Some loop paths do contain assignments that change the internal state, yet still allow repeating the same loop path. To detect changes in behavior after repeating this loop a certain number of times, we form a constraint that represents executing this loop multiple times. This constraint is created from the loop constraint. For a loop to be repeatable, the loop constraint that follows the loop path must still be satisfiable after going through the loop once. To verify this, we create an extended loop constraint. This extended loop constraint is simply a copy of the original loop constraint, however, variable indices need to be updated since some assignments already happened during the first loop.

For example, the loop constraint $a^i_0 > 1 \wedge a^k_1 = a^k_0 + 1$ represents a path with a branch, where variable i must be greater than 1 and the variable k is increased by 1. The extended loop constraint then becomes $a^i_2 > 1 \wedge a^k_2 = a^k_1 + 1$. For a given loop L we introduce the function σ_L, which performs this shift, s.t. $\sigma_L(a^k_i) = a^k_{i+t_k-s_k}$. Given a loop constraint P^0_L we extend $\sigma_L(P^0_L) = P^1_L$, where σ is applied to each variable appearing in P^0_L.

If the conjunction of the path constraint, the loop constraint and the extended loop constraint is satisfiable, the same path through the loop can be repeated once more. To create a loop constraint that represents executing this loop l times, we repeat the substitution process l times and check for satisfiability. We call this constraint obtained through the conjunction of repeatedly substituted loop constraints the unrolled loop constraint and it can be found in equation 1:

$$P^0_L \wedge \sigma_L(P^0_L) \wedge \sigma_L(P^1_L) \wedge \cdots \wedge \sigma_L(P^{l-1}_L) \tag{1}$$

Choosing the right value for l is however still a non-trivial decision. After continuing symbolic execution with the unrolled loop constraint added to the path constraint, only the right choice of l leads to generating inputs that trigger different branches. To solve this issue we add an iteration constraint. This iteration constraint allows further symbolic execution to use any number of iterations through the loop, up to the limit l. In the iteration constraint, all variables need to be equal to their respective symbolic values in one of the iterations through the loop. The iteration constraint is shown in equation 2:

$$\bigvee_{i=0}^{l}(n = i \wedge (\bigwedge_{\text{for all } j} a_f^j = a^j_{s_j+i*(t_j-s_j)})) \tag{2}$$

This creates a new index f for all variables, that represents their assignment after finishing the loop. For all variables, a_f^j is forced to adhere to one of the l assignments. When the SMT solver produces a satisfying assignment for some branch after the loop, it will need to fix n to a value between 0 and l which represents how many iterations of the loop were necessary to observe this behavior.

4.4. Obtaining counterexamples

If a loop is not a self-loop we want to generate a counterexample. Using our results from section 4.3.2 we will do this as follows: For each distinguishing trace of the hypothesis, its path constraint is collected by running 'access sequence · loop sequence · distinguishing sequence'. This results in the path the distinguishing trace takes after one iteration through the loop. The path constraint consists of two types of constraints. Constraints that originate from assignment statements and constraints that originate from branch conditions. These constraints are separated based on their types into two different constraints, the assignment constraint and the branch constraint. The branch constraint is then negated. The assignment constraint, the negated branch constraint and the path constraint created by running the loop once and adding the unrolled loop constraint get conjuncted together and given to the solver. If the solver finds a satisfiable model, the distinguishing trace follows a different path in one of the iterations. The trace then shows different behavior for this distinguishing trace after executing the loop a number of times[1]. The input that forms a counterexample for this loop is reconstructed from the model. The counterexample allows the learner to update the hypothesis. If none of the distinguishing traces yields a counterexample, the loop in the model is indistinguishable up to depth l. For each tested loop, this check calls the symbolic executor once for each distinguishing trace.

5. Experiments

We use the RERS challenges from 2020 for the experiments. The RERS challenge includes multiple problem categories, where the LTL problems lend themselves better for active learning. However, many of the LTL problems of 2020 do not require the techniques presented in this paper and can be learned with a W-method equivalence checkers with a shallow depth. This paper focuses on the reachability problems that require more sophisticated methods to find counterexamples of hypothesis models. We use problems 11 through 18 for symbolic execution[2] and problems 12, 13 and 15 for active learning to showcase the advantages of our methods. On all problems we don't specifically handle in these experiments all methods found the same amount of errors and counterexamples in little time and as such had equivalent performance.

1. Differences in the path does not necessarily mean different behavior. You can construct two paths with equivalent behavior. However, in our experiments this never occurred.
2. We were unable to run on Problem 16, as the instrumentation would cause the Java methods to exceed the maximum size. Splitting the methods could resolve the issue.

To verify our methodologies for symbolic execution and active learning, we set up three different experiments. In the experiments for symbolic execution, we only focus on getting more coverage, whereas in the active learning experiment, learning models is the goal. The entire source code we used in the experiments can be found at https://github.com/tudelft-cda-lab/SymLoop.

5.1. Symbolic execution

We built a symbolic execution engine called SymLoop that instruments Java source files to keep track of the symbolic values of each concrete variable. Z3 (de Moura and Bjørner, 2008) is used to answer any SMT queries. To speed up the symbolic execution, the executor evaluates expressions without symbolic values to constants (a technique also used by Klee). We use a simple heuristic to choose which input to run next. Any input which covers an unvisited branch takes precedence. If two inputs both cover new branches or both cover no new branches, shorter inputs are run first. Ties are handled by taking inputs that have been generated first. To compare our symbolic execution engine, we can also disable the loop detection and loop unrolling for a more traditional symbolic execution approach, we call this version the baseline. Additionally, we ran Klee for the same amount of time. During symbolic execution, we keep track of the errors found by each tool.

5.2. Model Learning

We implemented the Loop-W-method as an equivalence checker in the LearnLib framework (Merten et al., 2011; Raffelt et al., 2005). When the equivalence checker gets a new model to verify, it generates the set of distinguishing traces for the model. To generate the set of distinguishing traces, we use the existing functionality from LearnLib. The equivalence checker finds all loops in the hypothesis model. To reduce the number of loops to check, we only consider cycles, loops where only the first and last nodes of the loop are equal, while all the other nodes are unique. A Depth-first search (DFS) finds these circuits. After finding the loops, each loop is checked by concatenating the access sequence a with the loop sequence l and one of the distinguishing sequences d. This is repeated for each of the distinguishing sequences. If this does not yield a counterexample, the loop sequence l is added once more to get $a \cdot l \cdot l \cdot d$ and to check for equivalence. This process is repeated until a counterexample is found, or the specified depth limit l is reached.

For the symbolic equivalence checker, we implemented the methodology outlined in section 4.4. SymLoop collects the path constraint, checks for self-loops and generates the unrolled loop constraint before checking each distinguishing sequence. We ran the learning process using the three separate equivalence checkers. The standard W-method, the naive Loop-W-method and the symbolic method for verifying loops. For the learner, we used the TTT algorithm of LearnLib. The next section shows the results.

6. Results

6.1. Symbolic Execution

For the RERS Challenges, the results of running symbolic execution for 2 hours can be found in Table 1. On problems 11, 14, 17 and 18, there is no difference in the number of

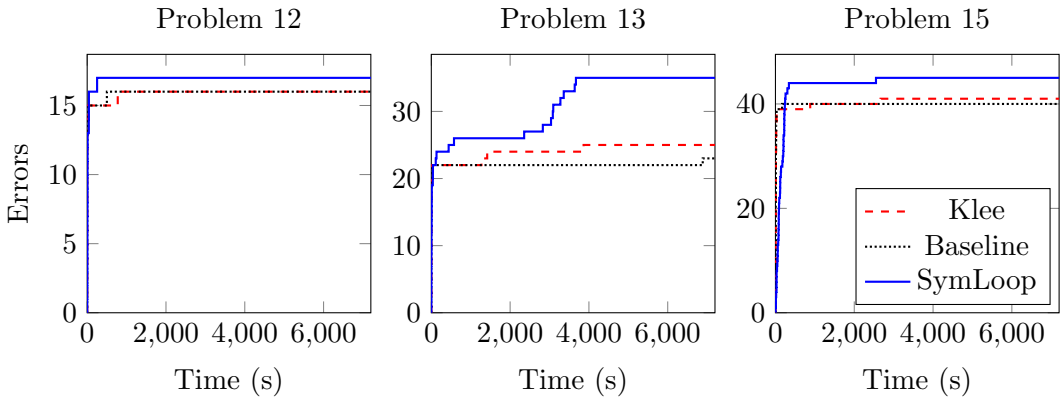


Figure 4: Number of errors found over time on the RERS problems. We have run all methods for 2 hours. SymLoop was run with a detection depth of 10 and a loop unroll amount of 50.

	P11	P12	P13	P14	P15	P17	P18
Baseline	18	16	23	15	40	30	30
Klee	18	16	25	15	41	30	30
LoopSym	18	**17**	**35**	15	**45**	30	30

Table 1: Results of running symbolic execution for 2 hours on the RERS problems 11 through 18, excluding 16. Each row represent a different symbolic executor. For each executor, the table shows the number of unique errors found per problem.

unique errors found. For problems 12, 13, and 15, there are differences. For these problems, we plotted the number of errors found over time in Figure 6.1. All methods quickly find the bulk of the errors, however, SymLoop is able to find more errors. On problem 12, the 17th error is found within 5 minutes, whereas Klee and the baseline do not find this error within the 2-hour time limit. Although SymLoop finds more errors, on both problems 13 and 15, the baseline and Klee are able to find an error that SymLoop did not find.

6.2. Model Learning

We have learned models for problems 12, 13, 15 and 17 of the RERS challenges. Our method stacks the W-method with depth 1, and the symbolic loop detector with a maximum loop size of 10 input symbols and an unrolled loop depth of 50. This is represented as the 'W1' for W-method with depth 1, and 'Symb D10L50' respectively. We have run the same experiment using a W-method with a W of 10 and the Loop-W-method. We let the methods run for 5 days before stopping them manually. Table 2 shows the number of states for each of the models. Due to the complexity of the problems, only problem 17 yielded an equivalent number of states for each method. On problems 12 and 13, the W-method did not find many states, whereas both SymLoop and Loop-W were able to find counterexamples for the

	P12	P13	P15	P17
W10	137	99	**507**	743
W1 + SymLoop D10L50	**11513**	**7125**	483	743
W1 + Loop-W D10L50	9838	2380	266	743

Table 2: Results of learning models of the RERS problems. Each row represent a different equivalence checker. For each equivalence checker, the table shows the number of states found per problem. The methods were run for 5 days, or until they finished.

hypothesis. As we have shown, loop equivalence checking can result in models with more states. An important observation is that the symbolic equivalence checker is only invoked a few times for each problem, and the preceding W-method with a depth of 1, found most counterexamples. When comparing SymLoop to Loop-W, SymLoop was able to find more states while requiring fewer membership queries.

7. Conclusions and future work

In this paper, we have provided methods for handling loops in both symbolic execution as well as for equivalence queries during active learning. We have also provided methods to integrate the handling of loops with symbolic execution into equivalence queries. This can be used to validate self-loops, something the W-method can not do. It is also useful when running membership queries is expensive as symbolic execution identifies counterexamples directly.

The results of our experiments show that on problems that contain such looping structures our methods outperform the state-of-the-art algorithm Klee in detecting errors during symbolic execution on the RERS challenge. The usage of the same symbolic execution methods has also been shown to be effective in finding more counterexamples during equivalence queries.

For cases where source code can't be accessed we have further adapted the W-Method to be more efficient in finding counterexamples for looping structures. Our work thus provides a diverse set of tools for handling looping structures.

For symbolic execution, we made assumptions on the execution model. This currently limits the type of code our method can be used for. We expect our methods to also work with a more general execution model, but leave this for future work. Another area where our method can be improved is handling counterexamples from loops differently from other counterexamples. For loop counterexamples, much of the previous behavior can be transferred to the updated model since we know that behavior only changes after a certain amount of iterations. For improved interpretability, these counterexamples are also suitable to be encoded as guards instead of unrolling the loop.

References

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987. ISSN 08905401. doi: 10.1016/0890-5401(87)90052-6. URL https://linkinghub.elsevier.com/retrieve/pii/0890540187900526.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51 (3):1–39, 2018. Publisher: ACM New York, NY, USA.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, and others. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978. ISSN 1939-3520. doi: 10.1109/TSE.1978.231496. Conference Name: IEEE Transactions on Software Engineering.

Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring openvpn state machines using protocol state fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 11–19. IEEE, 2018.

Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24.

Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, 2015.

Frank Drewes, Johanna Högberg, and Andreas Maletti. Mat learners for tree series: an abstract data type and two realizations. *Acta Informatica*, 48(3):165, 2011.

Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of dtls implementations using protocol state fuzzing. In *29th USENIX Security Symposium, Online, August 12–14, 2020*, pages 2523–2540, 2020.

Robert M. Hierons, Guy-Vincent Jourdan, Hasan Ural, and Husnu Yenigun. Using adaptive distinguishing sequences in checking sequence constructions. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 682–687, New York, NY, USA, March 2008. Association for Computing Machinery. ISBN 978-1-59593-753-7. doi: 10.1145/1363686.1363850. URL https://doi.org/10.1145/1363686.1363850.

Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen. The RERS challenge: towards controllable and scalable benchmark synthesis. *International Journal on Software Tools for Technology Transfer*, 23(6):917–930, 2021. Publisher: Springer.

Stefan Huster, Sebastian Burg, Hanno Eichelberger, Jo Laufenberg, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. Efficient Testing of Different Loop Paths. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 117–131, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22969-0. doi: 10.1007/978-3-319-22969-0_9.

Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*, pages 307–322. Springer, 2014.

Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon Schiffelers, Harco Kuppens, and Frits W. Vaandrager. RERS 2019: Combining Synthesis with Real-World Models. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 101–115, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17502-3. doi: 10.1007/978-3-030-17502-3_7.

Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. JDart: A Dynamic Symbolic Analysis Framework. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 442–459, Berlin, Heidelberg, 2016. Springer. ISBN 978-3-662-49674-9. doi: 10.1007/978-3-662-49674-9_26.

Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 220–223, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-19835-9. doi: 10.1007/978-3-642-19835-9_18.

Jan Obdržálek and Marek Trtík. Efficient Loop Navigation for Symbolic Execution. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 453–462, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24372-1. doi: 10.1007/978-3-642-24372-1_34.

Sebastian Poeplau and Aurélien Francillon. Symbolic execution with ${$symcc$}$: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

Harald Raffelt, Bernhard Steffen, and Therese Berg. LearnLib: a library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 62–71, New York, NY, USA, September 2005. Association for Computing Machinery. ISBN 978-1-59593-148-1. doi: 10.1145/1081180.1081189. URL https://doi.org/10.1145/1081180.1081189.

Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236, 2009.

Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. IEEE, 2017.

Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, pages 223–243. Springer, 2022.