Looping Structures in Symbolic Execution

Covering hard to reach code which requires many iterations through loops

Bram Verboom

TU Delft

Looping Structures in Symbolic Execution

Covering hard to reach code which requires many iterations through loops

by

Bram Verboom

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on April 21st, 2023

Student number:	4694384	
Project duration:	August 29, 2022 – April 21, 2023	
Thesis committee:	Dr. ir. T. E. Verwer,	TU Delft, supervisor
	Dr. T. Durieux,	TU Delft
	S. Dieck,	TU Delft, daily supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Software is everywhere, and going back to a life without software is unimaginable. Unfortunately, software does not always behave as expected, even though during the development cycle, software is usually tested to verify its correctness. To aid in testing, methods such as fuzzing or symbolic execution are used for automatic verification software systems. These methods are able to quickly find inputs to the systems that cover large portions of the code base. However, both of these methods struggle to find inputs that cover code which requires many iterations through loops.

In symbolic execution, loops are a large contributing factor to the path explosion problem and therefore the overall runtime. For loops containing conditional branches, each iteration is a new decision point. This leads to an exponential number of possible paths through a loop in comparison to the number of iterations.

In this work, we investigate the use of loops in symbolic execution to reach portions of the code which require numerous iterations through loops with conditional branches. We propose a novel technique for symbolic execution that uses the effects of one or more iterations through a loop to reach new parts of the code. By implementing this technique and applying it to a set of challenges designed to stress current tools and methods for software verification, we show that our technique is able to efficiently reach new parts of these challenges. These areas are not reached by state-of-the-art methods within the same time budget.

Another method for verifying software behavior is active learning, where simple models are learned from a system. These models capture the behavior of the system at a high level, allowing easier analysis to verify the behavior of a system. During the automatic learning of these models, loops are not handled separately. This leads to models where the behavior of a system is not captured fully, leading to incomplete analysis. We propose new methods for finding changes in behavior after executing these loops numerous times. We have compared our techniques to existing methods and show that this produces more complete models of a system.

Preface

Before you lies the result of 8 months of hard work. Many hours spent on reading papers, writing this thesis and building prototypes to explore concepts in the field of symbolic execution. I remember the first months of my thesis, not even knowing what to tackle, as so much in this field has already been explored. One day, I just ran some experiments to see what happened for a very small program. At that time, I was unaware of the impact those experiments would have on my thesis. This program is still present in the introduction of this thesis, and shows one of the problems in current symbolic executors. After five months, a handful of prototypes and many restless nights later, I managed to create something that worked. I cannot explain to you the joy and sense of accomplishment I felt when seeing the following string of characters:

> CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGI-
> HGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIB

After reading this thesis, you will hopefully understand how this repeating pattern shows that my hard work finally paid off.

This thesis would have been different without the support of many, and I am grateful for all the help. I want to thank my supervisor, Sicco, for the many fruitful discussions and your enthusiasm on the topic. Thank you Simon, for your criticism on my writing and for steering me towards progress. Thank you to the PhD students and other staff members that I could ask for help, but maybe just as important, the many times you got me to take a break and get some tea. Thank you to the other master students, for sharing courses and defending our table. For asking what I was working on and sharing your own work, for all the fun during and after the days of studying. You all made the commute to Delft worth it. Thank you to all my other friends, for convincing me to do a master in the first place, for bearing with me and for making the most out of the limited, yet much needed time away from studying. And last, but definitely not least, I want to thank my family for their massive support, not only during my thesis, but throughout all my years of studying.

Bram Verboom
Sunday 9th April, 2023

Contents

1 Introduction

The use of software for all kinds of purposes continues to increase. Our coffee machines are online, the lights in our offices turn on automatically when you walk in, and our hospitals keep your medical records in electronic patient dossiers. We become more reliant on all these systems, and a world without any software is hard to imagine. Although manufacturers make an effort to make their products function as intended, we have all probably experienced software systems crashing or failing. Some of these crashes can even be exploited for malicious intent. A common vulnerability is a buffer overflow, where an attacker can read or write data outside the bounds of the predefined buffer. Code reviews are common, but not everyone has the knowledge to spot the small mistakes that can compromise a system.

To aid in security testing, there exists methods such as fuzzing and symbolic execution that can automatically find crashes or failing systems. When a crash or bug is found in a system, the manufacturers can patch the system to prevent this from happening to end users. Fuzzers feed a program under test with random data to hopefully trigger crashes. With some additional nonrandom optimizations, this technique has been successful in finding bugs in many software systems. A limitation of fuzzing is the ability to reach code that requires more complex patterns or constraints on the input data. If you have the following condition in the code: `input == 1234567`, with 32-bit numbers as input, satisfying this condition randomly has a 1 in $2^{32} = 4294967296$ chance of occurring. With symbolic execution, finding inputs that make this condition true is trivial. The symbolic executor creates mathematical constraints for each condition and uses a constraint solver to generate an input that satisfies the constraints. Although symbolic execution can cover some parts of the code more quickly, the overhead of the constraint solver heavily impacts the overall execution speed. In most cases, a fuzzer can cover the majority of the code in a fraction of the time compared to a symbolic executor. However, fuzzers struggle with covering code that requires complex patterns in the input. We would like to find inputs that lead to crashes, so being able to reach code that does require many iterations through loops is desirable.

To illustrate how symbolic execution struggles with loops, we created an example program with a simple loop. The example C code is shown in Figure 1.1. This program takes in 2 inputs as command line arguments. The first input is a series of characters, and the second argument is a number that is stored in the `limit` variable. An initial variable `i` is initialized to `0` and will be updated when scanning the sequence of characters. The sequence of characters is scanned and checked one character at the time. If a character is an 'i', the variable `i` is increased by one, and the program will continue scanning the next character. The `i` variable therefore always contains the number of 'i' characters that that program has seen. If the character is a 'p', the `i` variable is checked. If the variable `i` is larger than or equal to the `limit`, the program will crash. If the limit is not reached yet, nothing happens and the next character is checked. When the character is neither an 'i' nor a 'p', the program will exit normally.

For symbolic execution, the sequence of characters is the symbolic input of the program. We set the limit to a specific value for one run of symbolic execution. By increasing the limit, we can test the ability of symbolic execution to reach code that requires more iterations through the loop. We have tested the runtime of two different symbolic execution tools, JDart and Klee. The C program

```
1  int main(int argc, char** args) {
2    assert(argc == 3);
3    int limit = atoi(args[1]); // Target to reach
4    int i = 0; // Internal state
5    int j = 0; // Loop variable
6    char symbol; // Character in input
7    char* trace = args[2]; // Input: array of symbols
8
9    while ((symbol = trace[j++]) != 0) { // Get next character in input
10     if (symbol == 'i') {
11       i += 1;
12     } else if (symbol == 'p') {
13       if (i >= limit){
14         assert(0); // Crash
15       }
16     } else {
17       return 0;
18     }
19   }
20   return 0;
21 }
```

Figure 1.1: C Program used for testing the runtime of Klee

used for Klee [3] is the code shown in Figure 1.1. The Java version required for running JDart [16] is semantically equivalent and left out.

The results of the runtime of the symbolic executors at different set limits are shown in Figure 1.2. To our surprise, the runtime grows exponentially in relation to the limit. This gave us the initial idea of improving symbolic execution by detecting looping structures.

These results can be explained by the path explosion problem in symbolic execution. In symbolic execution, every branch creates a new decision point for the symbolic executor, creating a new possible path to explore. To cover all the code of a program, all possible states of the program need to be stored, where each possible state takes a different path through the program. This rapidly growing number of paths is called the path explosion problem. In the example, each iteration of the loop allows three different paths; the input can be an 'i', the input can be a 'p' or the input can be something else. When the input is something else, the program exits. For an input of length n there are 3^n possible paths through the program. To find a specified `limit`, a breadth-first-search approach would require running 3^{limit} inputs to find the crash.

For this simple example, when knowing the underlying behavior, creating an input that crashes the system for a certain limit is relatively straightforward. An 'i' character in the input always increases the internal `i` variable by one, and the limit is compared to this `i` variable. Inputting a sequence of `limit` 'i's with one 'p' should always trigger the crash. This holds for an arbitrary limit, and generating this input does not take exponentially longer with an increasing limit. Instead, by reasoning over the program we can construct the example in linear time with respect to the limit, as the input is linear when compared to the limit.

1.1 Problem Statement

As illustrated by the example above, state-of-the-art symbolic execution fails to efficiently find inputs that cover code which requires a few iterations through a simple loop. We argue that if symbolic execution already fails for such a simple example, navigating code with more complex loops with more than just addition will also fail. Additionally, in normal programs, loops iterating for 10 or even 20 iterations occur frequently, and we expect that loops with even more iterations are not uncommon. As many programs contain loops, this is a major contributor to the path explosion in symbolic execution. The problem can be separated into two issues.

The first issue is the inability of current symbolic executors to detect that a path through the loop has no effect on the internal state of the program. To elaborate, we look at the simple example again. When a 'p' character is scanned before the limit is reached, no variables are changed, except the one that keeps track of the current position in the sequence of input characters. If the symbolic

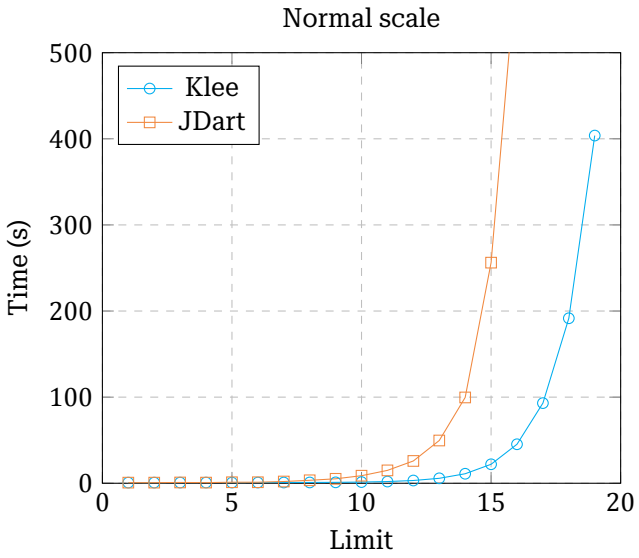


Figure 1.2: Time to run symbolic execution on program with a looping structure.

executor detects that none of the useful internal variables of the program have been changed, the symbolic executor could halt that execution, as there exists a shorter path that reaches the same internal state. For the example, this translates to stopping for inputs that contain a 'p' that does not trigger the crash. If the executor has two inputs in its queue: 'ii' and 'iip', they represent the same internal state, so the longest one with the 'p' could be pruned. Figure 1.3 illustrates the difference between pruning these self-loops and not pruning. This would already make the runtime for the example linear in relation to the limit.

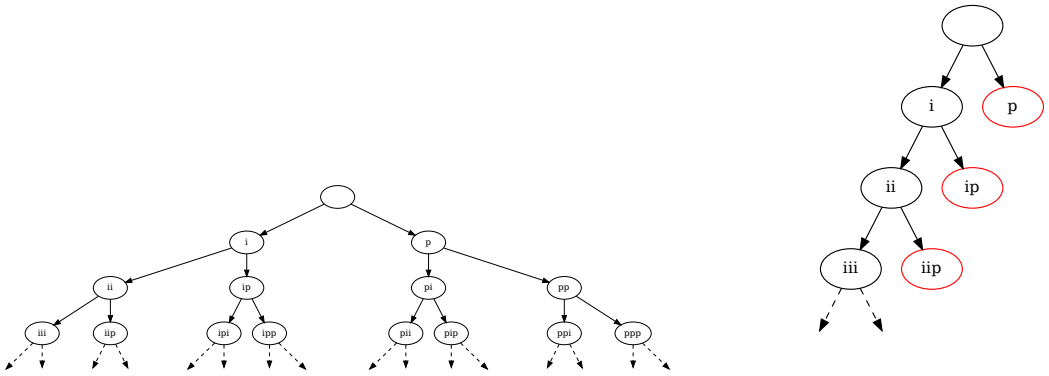


Figure 1.3: Illustration of the path explosion for self loops. The figure on the left shows all paths for current symbolic execution. The right figure shows the desired paths, where paths in red have equivalent internal states to their predecessor and thus does not have to be explored further.

The second issue is the inability of symbolic execution to consider the effects of a loop when executing the same path through the loop multiple times. By branching at each condition in the loop, the executor has an exponential runtime for the simple example. By analyzing the code, we can deduce that the `i` variable counts the number of 'i' characters in the input sequence. We can extrapolate the behavior of one iteration through the loop into a generalization of multiple iterations through that loop. To trigger the crash, the last character of the input needs to be a 'p' and the `i` variable needs to be larger than the limit. Therefore, to reach a certain limit, create a sequence of that many 'i' characters and one 'p' character.

In this work, we investigate the usage of loop detection and generalization strategies to aid symbolic execution in reaching code that requires more iterations through a loop. In publicly available state-of-the-art symbolic executors, no extra care is taken to analyze loops. Doing loop detection and loop analysis to create generalizations can lead to higher coverage, or more quickly being able to find certain crashes. We hope to create a new symbolic executor that has a sub-exponential runtime when running it on the simple example shown in Figure 1.1.

To keep the research objective feasible, we limit the scope to programs that have a similar input/output characteristic as shown in the example. Specifically, the input of a program must be a sequence of characters or integers and one iteration through the main loop of the program consumes one input of the input sequence. For our testing, we use the problems from the RERS challenge [15, 12, 11]. These problems adhere to the specific input/output behavior. The RERS challenge was designed to test the limits of analysis and verification techniques to improve or create new methods. A more in-depth explanation of the execution model can be found in Chapter 4.

There are several problems to be solved to be able to create a symbolic executor that can more efficiently navigate loops in the code. These problems form the basis of the subquestions of this research; these questions are shown in Section 1.2. The first major hurdle to overcome is the ability to detect the existence of a path through a loop which can be repeated. How can we do this for one iteration in the main loop? And can we do this with paths over multiple iterations through the main loop? Do we have to stick to the input/output characteristics as described above, or is it possible to do this for an arbitrary program?

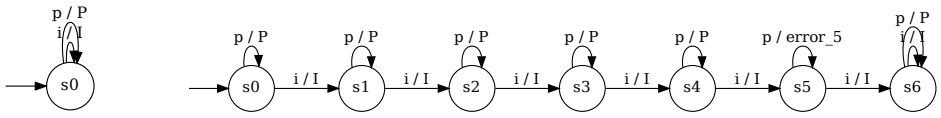
With a path that can be repeated, the effects of that loop path should be analyzed. Does that path change the internal state of the program? And if so, how? Ideally, we want to extrapolate the effects of one iteration of that path into a generalization that captures changes to the internal state when executing that path multiple times. In the example, the variable `i` was increased by one in each iteration in the loop. A generalization of adding 1 for n iterations is adding n once. We expect to be able to generalize this for some paths, such as the one from the example, but can we extend this to any path with any change to the internal state?

1.1.1 Motivation

With our simple example, we have shown that state-of-the-art symbolic executors struggle with the path explosion of loops. By analyzing the loop manually, it is relatively straightforward to create inputs that cover the code with the limit. Since many programs contain loops to process input, more efficiently handling these loops can benefit the runtime of symbolic execution for these programs. Since the runtime is currently exponential, a new method might even find new errors. In Chapter 3, we outline the research gap further. There is some research on this topic, but no technique is general enough to work for any program and loop structure. Some techniques rely on linear relationships between input and internal variables, sometimes even requiring an input grammar. Other methods rely on optimizable conditions and fail for branches which have conditions for which selecting inputs which are closer to triggering the branch does not lead to finding an input which satisfies the condition. The goal of this thesis is to create a general method which works for all loops.

1.1.2 Active Learning

In addition to extending symbolic execution, we also look at using symbolic execution for active learning. In active learning, a model is learned from a system by feeding the system with input and observing its behavior. Active learning is used to verify the functionality of the software. After learning a model, analysts can compare the models against expected behavior. The analysis of models that were learned from software has been successfully used to detect bugs in real implementations of TLS protocols by De Ruiter and Poll. In active learning, there is no distinction made between transitions through states that form cycles in the model and transitions that are not part of a cycle. Creating a distinction between those two sets of transitions can allow checking whether the cycles in the loop are also cycles in the system from which the model is learned. We investigate whether the cycles in the model show different behavior after a number of iterations of that loop. Identifying such a change results in many new states and thus aid the active learning process, see Figure 1.4.



(a) Change not found

(b) Change in loop behavior found

Figure 1.4: 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 that results in the right model does capture this behavior.

1.2 Research Questions

The main objective of this research is to extend symbolic execution with methods that allow more efficient traversal of loops in the program. Hopefully, this can reduce the path explosion problem for the instances where we consider current methods to be inefficient.

How can you extend symbolic execution to reach states that require repetitive iterations through loops?

Answering the following sub-questions allows us to create an extension that allows symbolic execution to reach code which requires several iterations through a loop.
RQ1: How can loop structures in symbolic execution be detected?
RQ2: Is it possible to generalize the assignments in loop structures into constraints for a symbolic executor?
RQ3: How can loop structures be used to reach new branches in symbolic execution?
RQ4: Does detecting and using loop structures allow symbolic execution to cover more states of a program?
RQ5: How can we use symbolic execution for equivalence oracles in active learning?

1.3 Contributions

In this thesis, we propose a new technique for symbolic execution that allows covering code that requires many iterations through a loop. Some existing methods only work on linear relationships between the input and some of the internal variables of the program. Some methods even require that the input grammar of a program be specified. Our method is able to handle arbitrary updates to the internal state of the program without needing an input grammar.

We implemented our symbolic executor and applied it to the RERS Challenge [15, 12, 11]. Our approach outperforms state-of-the-art symbolic execution and fuzzing by finding more errors within the same time window. In several cases, our symbolic executor can find errors in 5 minutes, which the state-of-the-art symbolic executor Klee could not find in 24 hours. However, loop detection comes with a runtime cost, which is why Klee found some errors which our methodology did not find in the same time frame.

In addition to our symbolic executor, we also developed and implemented new methods for equivalence checking of loops during active learning. We introduce two different methods:

- Our naive Loop-W-method works in all cases and is able to detect changes in loop behavior. These changes can not efficiently be detected in traditional equivalence checking techniques such as the W-method.
- Our symbolic method uses the concepts from the symbolic execution engine to create constraints to check the behavior of a loop path after many iterations. The symbolic method requires more information from the underlying system, but has the benefit of requiring fewer membership queries to get the same result as our naive method. Our symbolic method can also guarantee that some of the cycles in a hypothesis model are correct.

When using our methods as equivalence checkers for learning models of the RERS problems, both are able to find significantly more states than the traditional W-method equivalence checker.

We have also condensed this thesis into a research paper that was submitted to the 16th International Conference on Grammatical Inference. The paper has been included in this report as Appendix B. At the time of writing, we have not received a notification of acceptance, as the review process is still ongoing.

1.4 Outline

The next chapter introduces the necessary background information on symbolic execution, as well as the concepts and methods for active learning. Chapter 2 also shows related work to address the problems defined in this introduction. The chapter following the related work outlines the issues with these methods and positions our approach in terms of the limitations of the existing approaches. In Chapter 4, we provide the execution model that is used throughout this research. Chapter 5 answers the first research question on detecting loop structures. Chapter 6 uses the loop structures to answer the next research question on generalizing the effects of these structures. The next chapter uses these generalizations to extend symbolic execution, with Chapter 8 applying these methods in the experiments. The following chapter, Chapter 9, answers the research question of using symbolic execution in the context of active learning. All results are shown in Chapter 10 including the analysis of the results. Chapter 11 provides the final conclusion and shows future work.

2

Background and Related Work

2.1 Automated Vulnerability Discovery

AVD (Automated Vulnerability Discovery) is the process of automatically finding bugs, crashes, or other faults in programs. Numerous techniques were developed over decades of research in this area. The techniques vary from random fuzzing: providing random input to a program and detecting if the program crashes, to more advanced techniques like symbolic execution that use the internals of programs to reason about possible crashes.

2.1.1 White-, Grey- and Black-box

Techniques for AVD can be categorized based on several criteria. One of the criteria we are interested in is the level of knowledge required of the system. We outline the main differences below.

White-box methods are sometimes also called glass- or crystal-box methods, and just like the contents of a glass box, the internals of these systems are fully visible. All intermediate steps for producing the output are visible. For software systems, seeing their internals translates to viewing their source code. White-box techniques also apply when inspecting compiled binaries. Although some details are lost in the compilation process, the functionality is still embedded in the binary.

Black-box methods consider a system to be a black box: after being given an input, it produces some output. How the system generates its result based on the input is completely hidden. Due to the limited knowledge of a system, black-box techniques are less powerful when compared to white-box methods.

Grey-box methods are a middle ground between white- and black-box methods. Although there is no general definition for grey-box, for these methods only limited information is available.

Another distinction is the notion of static or dynamic analysis. For static techniques, the analyzed programs or systems are never run. In situations where running a system and observing its output is costly, static techniques can provide useful insights. Static analysis analyzes the source code or binary. In dynamic techniques, the programs or systems are run to observe their behavior. During runtime, the internal state of the system can be inspected. This allows dynamic techniques to analyze different properties.

2.2 Types of Faults

Systems can contain various types of faults. AVD often focuses on bugs that lead to crashes, as a crash of a system is easy to observe. Crashes can be caused by reading or writing to buffers past their bounds, reading uninitialized memory, failing assertions, or other mistakes in developing the system. Behavioral bugs or mistakes are not caught by vulnerability discovery, to illustrate: a system that is meant to add numbers but instead multiplies them is behaviorally wrong. However, the code does not contain any bugs that could trigger it to crash.

2.3 Fuzzing

There exists several techniques for finding vulnerabilities. One of these is fuzzing. In fuzzing, a program or system is repeatedly provided with different inputs. By repeatedly providing different inputs, a system might crash for one of the inputs. Dumb fuzzers can just try random inputs, but more intelligent methods use coverage to guide the fuzzing process. AFL++[9] is one of the state-of-the-art coverage guided fuzzers. AFL++ uses mutators to cover new areas of the system.

2.4 SAT and SMT Solver

The satisfiability problem, also known as SAT, is the problem of deciding whether a propositional formula can be made true by an assignment of the variables in the formula. The problem is NP-Complete, so finding a solution in polynomial time is believed to be impossible. If the formulas extend to integers and strings, instead of just boolean operations, the problem is called SMT (Satisfiability Modulo Theory). Especially for use in symbolic execution (see Section 2.5), being able to use and solve mathematical formulas over numbers or strings is required, as most programs also perform arithmetic on numbers or operate on strings. Formulas in propositional logic are a strict subset of SMT formulas, so computing satisfiability for mathematical formulas can not be easier then computing satisfiability for problems in propositional logic. Since the SAT problem is NP-Complete, the SMT problem is also NP-Complete, therefore finding a solution in polynomial time is also believed to be impossible.

There exist SMT-solvers that can solve mathematical formulas in non-polynomial time, and a lot of effort has been put into making them faster and more efficient. An example of such a solver is **Z3** [18]. During this report, we will use the terms solver and SMT-solver interchangeably for a program that solves a formula in modulo theory and indicates that it is unsatisfiable or satisfiable. Additionally, if the formula is satisfiable, it gives back a model that satisfies the formula and includes the assignments for each variable in the formula.

2.5 Symbolic Execution

Symbolic execution is a technique to find inputs for a program that traverse different paths in the program. It can be used to detect bugs, but also to generate test suites that cover different behaviors in the program under test. Instead of running a program normally, symbolic execution executes the program abstractly and reasons about multiple possible inputs by leaving the inputs to a program as free variables, creating symbolic expressions for internal variables, and then using a solver to discover new paths in the program.

For this work, we will focus on dynamic symbolic execution or also called concrete symbolic execution or concolic execution. This section will provide the reader with a basic understanding of concolic execution.

Throughout the explanation of symbolic execution, we will use an example of running symbolic execution for the program found in Figure 2.1. This program takes in an age as input. The program then outputs whether the age is invalid, it is the age of a child, or it is the age of an adult.

2.5.1 Variable Assignment

When starting concolic execution, the program under test is run on a sample input. During the execution, any changes to variables are recorded, and a path constraint is formed that matches the expressions that are evaluated. Any input to the program is kept as free variables. For the example, the initial assignment is `age = input()`. This creates the symbolic expression '$age_0 = input_0$', where '$input_0$' is the free variable representing the input to the program. If any variable is updated again, a new constraint is formed which assigns the new value of that variable. Every time an assignment occurs, the symbolic executor will create a new symbolic variable that corresponds to the current value of that variable in the code. These assignments follow the principle of SSA (Static Single Assignment).

2.5.2 Conditional Statements

Whenever conditional statements are executed, the condition is evaluated and before continuing to execute down that path, the following occurs: if the condition is satisfied (i.e. `true`), the negated

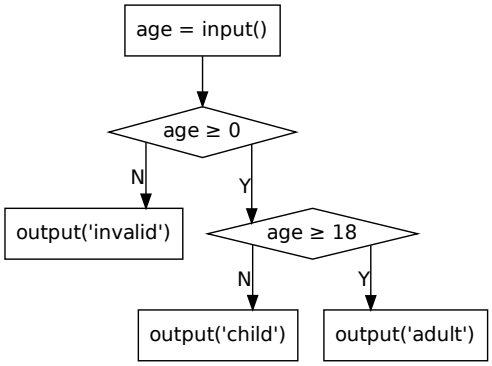


Figure 2.1: Example program for explaining symbolic execution. The program is represented as a CFG

version of that condition is created and given to the SMT solver alongside the current path constraint. Afterward, the path constraint is updated such that the condition must be satisfied. Whenever the condition is not satisfied (i.e. `false`), the solver is given the condition along with the path constraint. Afterward, the negated condition is added to the path constraint. In both of these cases, the solver is executed and if it can satisfy the given constraints, a new input to the program can be formed and gets added to a queue of inputs. The model given back by the solver will contain concrete values for each symbolic input. The goal of using the solver is to find an input that executes the opposite branch of the conditional statement. Throughout the report, we specifically chose to use condition to refer to the expression that is contained in a conditional (i.e. `if`) statement, where that condition decides which branch to take. When using the word constraint, we refer to a logical formula that is used in the context of SMT solvers.

When looking at the example, if the initial input -1 was provided as age, the condition `age ≥ 0` would be `false` and execution would follow the `N` path. To find an input which reaches the other path, the `Y` path, the condition is converted to the constraint $age_0 \geq 0$. This condition is conjuncted with the current path constraint and given to the solver: $(age_0 = input_0) \wedge (age_0 \geq 0)$, the solver then tries to find an assignment to the variables which satisfies the constraint. A possible assignment to the input would be $input_0 = 1$, so by inputting an age of 1, the `Y` path would be followed. After solving for the current branch and adding the new input to the queue of inputs, the path constraint is extended with the negated version of the condition (remember, the condition is `false`, yielding the path constraint $(age_0 = input_0) \wedge \neg(age_0 \geq 0)$. After adding this to the path constraint, the execution is continued to the next statement, in this case outputting 'invalid' and exiting.

2.5.3 Input Queue

After running the program on the initial input, a new input is retrieved from the queue. All the inputs in the queue follow a different path than the one used to construct that input. The input is retrieved based on a heuristic, where each concolic execution engine uses a different heuristic or allows the user to specify one of its built-in heuristics. A problem in symbolic execution is the quickly growing queue of inputs, which may contain duplicates, so a strategy to remove redundant inputs is beneficial to keep path explosion to a minimum.

For our example, the initial input -1 produced the input 1, so the queue just consists of the input 1. The input 1 visits the `age ≥ 0` conditional statement, which generates the input -1. After following the `Y` path, it also visits the conditional statement `age ≥ 18`. This generates a new input using the constraint $(age_0 = input_0) \wedge (age_0 \geq 0) \wedge (age_0 \geq 18)$. An input that satisfies this condition is 20. Since the input -1 is already ran, only the input 20 is run, which leads to all branches being covered.

2.5.4 Path Explosion
Path explosion refers to the fact that the number of paths through a program grows exponentially in terms of the length of the input or the number of conditions in a program. In theory, at every condition a different path can be followed, leading to a decision point at every conditional statement. Although in practice not all paths can be followed due to the constraints being unsatisfiable, the number of paths still grows rapidly. Loops are a large contributor because, after each iteration of a loop, a decision is made whether to exit the loop or continue another iteration. Additionally, every conditional statement in a loop is another decision point, so for a loop that iterates n times, this condition creates 2^n additional decision points.

2.5.5 Source Code, Intermediate Representation or Binary
Concolic execution can be made possible through different methods, some methods require the source code of the program under test to instrument the program and add the code necessary for the symbolic reasoning. Other methods work on an intermediate representation that a compiler produces (commonly LLVM IR) to support more programming languages that produce this intermediate representation. Some methods work on already compiled binary programs, for example, if the source code is not (publicly) available.

2.5.6 External environment
A hard problem in symbolic execution is the use of external environments, such as the file system, the network, external libraries, or other programs. Symbolically representing the interactions between the environment and the program under test can be done by simulating these interactions, symbolically interpreting the libraries, or in other cases where either of those is not possible, executing them normally and using the concrete values of these interactions.

2.5.7 Advancements in Solvers
A large contributing factor of the runtime of symbolic execution is the overhead of the solver. At every branching point in the program, the solver needs to invoked with the current path constraint. Due to the heavy usage of the solver, symbolic execution automatically benefits from advances in SMT solvers.

2.6 Automata Learning
Previously mentioned techniques focus on automatically finding errors that exist in the implemented versions of the program (programmatic errors). Not all incorrect behavior causes immediate crashes or errors. A coffee machine that allows you to get free coffee whenever you enter a specific code is still functional, yet this behavior is still problematic. Models of systems allow analysts to find logical errors in the system. For the coffee machine, a path can be found where the `brew coffee` state could be reached without ever going through a `payment` state.

2.6.1 Definitions
This section contains some common definitions we will use throughout this paper.

State Machine
State machines are a common analogy to regular languages, as any regular language can be built as a state machine that can decide whether a word (a series of input symbols) is a member of the language. The state machines we use are DFA (Deterministic finite automaton). A state machine $\mathcal{D} = (Q, F, \Sigma, \Gamma)$ is a 4-tuple consisting of:

- A finite set of states $Q = \{q_0, q_2, \dots q_n\}$ where q_0 is the initial state,
- A mapping from state to accepting or rejecting $F : Q \to \{+, -\}$,
- An finite set of input symbols Σ
- A transition function mapping state and input to a new state: $\Gamma : Q, \Sigma \to Q$

Mealy Machines

Mealy machines are an extension of the state machines introduced in the previous paragraph. Mealy Machines also produce an output upon transitioning to a state when consuming an input symbol. Because of this extra behavior, Mealy Machines are 5-tuples $\mathcal{M} = (Q, F, \Sigma_i, \Sigma_o, \Gamma)$ consisting of:

- A finite set of states $Q = \{q_0, q_1, \ldots q_n\}$ where q_0 is the initial state,
- A mapping from state to accepting or rejecting $F : Q \to \{+, -\}$,
- An finite set of input symbols Σ_i
- An finite set of output symbols Σ_0
- A transition function mapping state and input to a new state and the output it produces: $\Gamma : Q, \Sigma_i \to (Q, \Sigma_o)$

[1] Proposed the MAT (minimally adequate Teacher) framework consisting of a Teacher and a Learner interacting to let the Learner form a description of a regular set. A minimally adequate Teacher should correctly answer two types of questions from the Learner, the simplest one of them is the membership query and the other question is a conjecture. With these two types of questions, a state machine can be constructed.

Membership Query

The student can ask membership queries to find out whether a sequence of symbols is a member of the regular set. The Teacher then responds either with a yes or no. Typically, the teacher runs the sequence of symbols through the program under test or against the regular set. The runtime of this query is bounded in terms of runtime by the length n of the sequence of symbols $O(n)$.

Conjecture

A conjecture is a hypothesis that the learner has of the regular set. The learner asks the teacher if the hypothesis is correct. The hypothesis can either be correct or the teacher returns a counterexample. This type of question is in practice a lot harder because the teacher also does not have full knowledge of the regular set.

2.6.2 L* Learning Algorithm

We give a high-level overview of the L* (pronounce: L-star) learning algorithm that was introduced in [1]. The L* learning algorithm abides by the minimally adequate teacher framework to actively learn state machines of regular languages. The L* learner uses an observation table to keep track of prefixes and suffixes along with their membership in the set. The observation table (S, E, T) consists of a nonempty set of prefixes S, a nonempty set of suffixes E and a membership function T mapping 'prefix · suffix' and 'prefix · input symbol · suffix' to non-membership or membership ($\bot$ and $\top$ respectively) $T : (S \cup (S \cdot \Sigma)) \cdot E \to \{\top, \bot\}$. The suffixes are along the rows and the prefixes are along the columns. The prefix and suffix sets start with the empty word $S = E = \{\varepsilon\}$. The learner then asks the teacher membership queries to fill out the observation table. This process will continue until the observation table is filled. Whenever the observation table is filled, the learner checks whether the table is closed and consistent. The observation table is closed if it can create transitions for each unique state for all the symbols in the input alphabet. The observation table is consistent if all rows in the observation table that correspond to the same state lead to the same states for every symbol in the input alphabet. Once the observation table is closed and consistent, a hypothesis model can be sent to the teacher. This hypothesis model is checked for equivalence. If the equivalence checker finds a counterexample, for which the model does not match the system under test, the observation table must be updated to reflect this counterexample. The original paper by Angluin suggested adding all prefix of the counterexample to the set S. Although there are methods to reduce the counterexample to only add minimal information to the observation table, the approach of adding all prefixes of the counterexample works.

2.6.3 TTT

The TTT algorithm [14] is also an active learning algorithm for constructing DFAs. The goal of creating TTT was to reduce the number of membership queries and make the observation tables less redundant than the methods introduced before the TTT paper. To best explain the TTT algorithm, we will introduce some additional notation.

- Σ^*: All finite words made from combining symbols in Σ, including the empty word ε.
- Σ^+: All finite words made from combining symbols in Σ, excluding the empty word ε.
- $\lambda_q : \Sigma_* \to \{\top, \bot\}$ of q with $\lambda_q(v) = T$ iff $F(\Gamma(q, v)) = \top$

The TTT algorithm represents the observations in a different structure than the table of L*. TTT uses a discrimination tree. This discrimination tree distinguishes between the states based on prefixes. Due to the representation being redundancy-free, TTT can learn the same model with fewer membership queries, while also having a lower space requirement [14].

2.7 Answering Conjectures

After a learning algorithm has created a hypothesis automaton based on its observations, a conjecture needs to be answered to verify that the model is equivalent to the system under test.

A naive way to verify a hypothesis is to brute force all possible words up to a certain length. The membership queries of running a word through the system is compared to the result of running the same words through the hypothesis. If the membership indicates that the word is contained in the language, but the hypothesis does not agree (or vice versa), this word forms a counterexample and is given back to the learner to refine its hypothesis. This method guarantees that for all words up to a certain length, the hypothesis is equivalent. However, this approach can be infeasible due to the exponential complexity in terms of the length and alphabet size.

2.7.1 Distinguishing Sequences

A distinguishing sequence is an input sequence which distinguishes two states according to the observed output of running this sequence. For a deterministic state or mealy machine are a set of distinguishing sequences is the set which distinguishes all states from each other. This set has the same size as the number of states in a model[1].

2.7.2 W-method

One method for equivalence checking is the W-method. The W-method [7] with specified parameter w can guarantee there does not exist a state machine with less than w states more then the current hypothesis. The W-method returns counterexamples as long as it can find behavior which does not match the current hypothesis. If the W-method does not return a counterexample, the model is correct or needs at least w more states. In contrast with the brute-force approach, this method is more efficient and uses a distinguishing set to be able to deduce in which state the system is in. The W-method checks all words consisting of an access sequence A, a word with at most w symbols of the input alphabet, and a distinguishing sequence. The access sequences and distinguishing sequences are created from the hypothesis model. The access sequences are a set of words which each lead to a different state in the hypothesis model. If the output of running any of these words through the system is different to the output of running the same word through the hypothesis model, a counterexample is found. The hypothesis then needs to be refined to correctly capture this behavior.

As an alternative to checking all the words with at most w symbols, there are alternative methods which perform better on average, but have less strong performance guarantees. One of these methods is the rrandom-W-Method. The random-W-method randomly chooses an access sequence, a word of at most w input symbols, and a random distinguishes sequence. These are concatenated and checked. This process is then repeated a large number of times.

[1]For a mealy machine, this does not have to hold in all cases, but it holds for the worst case.

2.7.3 L* for Mealy Machines

A variation on the original L* algorithm LM* was specially created by Shahbaz and Groz for Mealy Machines. The observation table is modified to include output strings instead of just recording '1' or '0' [23]. The suffix set E is also altered to initially include all the symbols of the input alphabet. The rest of the algorithm still works and is kept as-is.

2.8 RERS Challenge

Since 2010, the RERS Challenge was held to test and improve software verification tools. In the Rigorous Examination of Reactive System challenge, in each edition, a set of programs is published, and some questions are asked about these programs. The RERS challenge has been created to test the limits of software verification tools, as well as to expand the framework used for generating the challenges themselves.

The programs in the RERS challenge are reactive systems: they repeatedly consume inputs from a predefined set of input symbols and output symbols from a predefined alphabet. When processing the input, the systems can also crash with a specific error code. The source code for each program is available. There are several programs divided into two different categories. In the reachability category, the aim is to be able to indicate for each error in the code whether it is reachable by some input. The other category is the set of LTL problems, for this category, each problem gets accompanied by a set of LTL (Linear temporal logic) formulas. For each of these formulas, a challenger needs to output whether the formula is true or false. Due to it being a more complex problem, we have considered this category to be outside the scope of the thesis.

2.9 Loop Summarization

Tangent to this research is loop summarization, to simplify loops to be able to execute them faster or analyze termination properties. In compilers, loop summarization is usually implemented to reduce the runtime of a program. An example of loop summarization in action can be seen in the compilation of the following program:

```
int with_loop() {
    int res = 0;
    for(int i = 0; i < num; i++) {
        res += a;
    }
    return res;
}
```

For this program, the x86-64 gcc compiler with `-O3` optimization flag generates the following assembly:

```
with_loop(int, int):
        test    edi, edi
        jle     .L7
        mov     eax, edi
        imul    eax, esi
        ret
.L7:
        xor     eax, eax
        ret
```

In this code, the `for` loop is summarized as a multiplication. The following C code generates nearly the same assembly, with only a different order. It shows that the compiler is smart enough to rewrite repeated addition into a semantically equivalent multiplication.

```
int without_loop(int num, int a) {
    if (num <= 0) {
        return 0;
```

```
    }
    return num * a;
}
```

2.10 Related Work

For active learning, we could not find research specifically looking at the verification of the loops in the models. This seeming gap in research indicates a new area to explore and marks the importance of the research question on using loop detection for equivalence checking.

Other relevant and recent work on active learning or model learning is listed below.

MACE [5] uses symbolic execution to form a model of the program, where the model can then again be used in symbolic execution. This continuous process allows better exploration of the program, due to the constantly improving approximation of the program.

Using **Adaptive distinguishing sequences** (ADTs) [10], 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 learn is in the expected state. Additionally, there exists models for which a preset distinguishing set cannot be found.

2.10.1 Model Learning with fuzzing

Fuzzing has also been applied to do model learning [6]. By fuzzing the system under learn and capturing its input-output data, these traces can be used as a corpus to verify the model against. If the outputs in the traces generated by the fuzzer does not match the output of a hypothesis model, a counterexample is found and is given back to the teacher. In this work, there is no link back to using the model to better guide the fuzzer.

2.10.2 Model Learning by Apartness

Recent work by Vaandrager et al. 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 focussing 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 method such as TTT by needing a comparable number of membership queries. Their method does however outperform state-of-the-art algorithms by requiring fewer number of symbols for learning.

This section shows existing techniques in symbolic execution to reduce the execution speed or reduce the path explosion problem to be able to find errors requiring numerous through loop.

Klee [3] is a concolic execution tool developed by researchers who already had experience with building symbolic execution tools beforehand [4]. Klee uses several techniques to improve the runtime of the symbolic execution to reach more of the program. Most of the optimizations focus on minimizing the number of calls to the solver. “Almost always, the cost of constraint solving dominates everything else” [3]. Several of these techniques used for reducing the solver cost are listed below.

- Rewrite expressions to simpler forms, such as simplifying $x + 0 = 0$ to $x = 0$. This is much like compiler optimizations for simplifying expressions.
- Simplify constraint sets to remove redundant statements, such as $x > 1 \wedge x = 10$ to $x = 10$, since one constraint implies the other one.
- Concretize implied values whenever Klee detects that a value can only take on one specific value, any subsequent accesses will use the concrete value instead of its symbolic value.
- Divide constraints into independent disjoint subsets to eliminate constraints that are irrelevant to the query. For example, the constraint set $\{x > 10, y < 1\}$ with the query $x = 10$ can be minimized to $\{x > 10\}$ since the y variable is irrelevant.
- Use a counter-example cache to skip the solver. When a subset of constraints is already unsatisfiable, adding more constraints cannot make it satisfiable, therefore the query can be skipped.

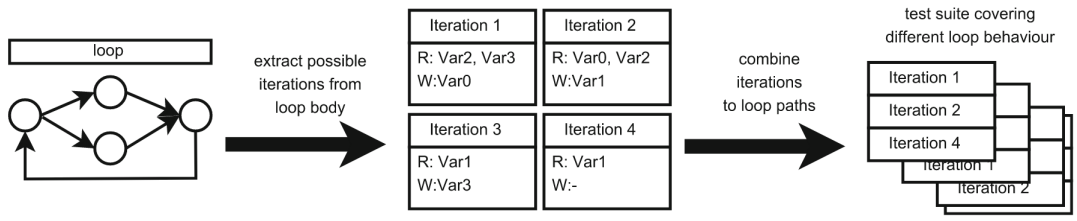


Figure 2.2: High level overview of the methodology of Efficient Testing of Different Loop Paths. The image is taken from the original paper [13].

SymCC [20] is another symbolic execution engine that was build to minimize execution speed. The main speed benefits rely on compiling the symbolic execution right into the same program. Engines like Klee are based more on interpretation, whereas SymCC is built as a compiler for C and C++. The LLVM-based compiler is able to inject all the necessary code for symbolic execution. During runtime, no time then needs to be spent on interpreting the code that is currently being executed. In their paper, they state an average speedup of a factor of 12 when comparing against Klee.

Although both SymCC and Klee improve the overall speed of symbolic execution, they do not improve the overall search method. Therefore, when handling loops, both still struggle with the path explosion problem. The next sections list relevant literature that tackles the path explosion problem caused by loops.

Loop Extended Symbolic Execution (LESE) was published in 2009 [22] and has since then become one of the more cited papers in this area of research. Their work introduces the symbolic variables for the number of times each loop is executed. These are linked to the patterns in a predefined input grammar. Traditionally, symbolic execution creates a path constraint representing a single path that is executed. LESE creates symbolic constraints for variables that represent multiple paths through the loop. In their work, they look for linear relationship: 'Specifically, our tool searches for variables whose value is a linear function of trip count variables representing the number of times one or more loops execute.' This is also a limitation of the work, since non-linear relationships can not be learned.

Efficient Testing of Different Loop Paths by Huster et al. introduced a methodology for analyzing multiple different paths through a loop [13]. By leveraging static analysis, possible loop paths are extracted from the program. Each loop path or iteration includes reads or writes to different variables. The different iterations are combined to cover different behavior of the loop. If one path only reads from `Var1` and modifies `Var2`, and another path only reads `Var3` and writes to `Var4`, the execution order of these iterations does not influence the final result. By analyzing the read and writes for each iteration, they can create combinations of these iterations that affect each other. An overview of the process is shown in Figure 2.2. The combinations are generated based on the weights of basic blocks in the loops' path. The weights indicate how often a basic block needs to be covered. Based on weights, possible loop executions are generated that cover a set of basic blocks.

Efficient Loop Navigation for Symbolic Execution [19] is another method that tackles the same problem. Our example shown in Chapter 1 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 which require more loop iterations.

Veritesting [2] uses state merging to reduce the number of paths. The states are merged when they have similar values for any internal variables. The values are conjuncted together in the path constraint. The paper mentions that boolean operators are however not allowed in sub-expressions. Additionally, their results show that their method still scales exponentially for an increasing number of iterations [24].

3

Research Gap

As we have already concluded in the chapter on background and related work, we could not find any research for active learning that specifically checks loops in models. This already shows the research gap on this topic. In this chapter, we describe our approach for developing a method that allows symbolic execution to reach codes which requires many iterations through a loop. Our approach is formed from the limitations in existing methods.

3.1 Issues of Existing Approaches

To create our approach, we describe the limitations of current approaches. We have based these issues on other methods from literature that tackle similar problems. Some methods have only one limitation, where other methods have multiple of the issues described below.

Breadth first search With current state-of-the-art executors that are available to the public, loops are not handled separately. When compiling source code to machine-readable instructions, some of the information on where the loops are is lost. In machine code, or some form of intermediate representation, the notion of a loop is just a series of statements with a (conditional) jump to a previously executed block. The traditional way of symbolic execution is to do a breadth-first-search. Whenever a conditional jump occurs where both branches are satisfiable, the state is split into two separate states. The symbolic executor alternates further execution of these two states. When a new conditional branch occurs in a state, that state is again split, and the symbolic executor divides its resources over all the discovered states.

There are heuristics to prioritize exploring some states and skipping others. For example, by default Klee alternates between random exploration and exploring states with new coverage. This is not a breadth-first-search, but this heuristics still fails for our example program in the introduction.

Restricted to linear relationships The problem for the example shown in the introduction has also been mentioned in other literature. We wanted to test the proposed methods to solve that problem, but we did not find any source code to run it on the example or other programs. The example we showed has one aspect which most literature focuses on, there is a linear relationship between the internal variable `i` and inputting an extra 'i' character. However, there is not always a linear relationship between the input of the program and the internal state after processing that input. A simple example for this is multiplication. If we had a program that is similar to one shown in the introduction, but instead of adding `1` to the `i` variable, the variable is multiplied by `2`. This is no longer a linear relationship, but the transformation leads to an exponential relationship between the number of 'i' characters in the input and the value of the `i` variable. Since a program can contain any expression, detecting linear relationships only solves a small portion of the program.

Non-Optimizable conditions Some of the techniques mentioned in Chapter 2 use optimization strategies to find inputs which are closer to satisfying a specific conditions. Again, we refer to the example from the introduction. In this example, adding an extra 'i' to the input brings the internal state numerically closer to the condition which contains the limit. The branch distance of a specific condition (or branch) can be computed by taking the expression and computing how close the current evaluation is to flipping that condition. If the current evaluation of a condition is $1 > 10$,

which is currently false, making this condition true would require the 1 to be changed to at least 11, or the 10 to at most 0 leading to a distance to 10. For the example from the introduction, the branch distance can be used as a heuristic for generating inputs that would reach a limit. However, this technique does not work for any condition or update to the internal variables. To illustrate, consider the modulo operator. Taking the modulo 100 of an increasing variable only increases the result at first, but when the variable reaches 100, the value suddenly becomes 0 again. This behavior is hard to optimize for, as the value increases before it reaches its lowest value.

Require an input grammar Existing methods such as LESE[22] require an input grammar to be able to learn relationships between the input of a program and its internal state. Although this can be a powerful method, manually defining the input grammar of a program can be time-consuming. Additionally, when a part of the input grammar is missing or incorrect, the relationship can still not be deduced. In some scenarios, like reverse engineering, the behavior of the program to be tested is unknown. Creating a complete input grammar for these programs becomes hard, if not impossible, to do. Since the behavior of the program is already embedded in the binary or source code, we want to use this information and skip the additional step.

3.2 Position of Our Approach

Guided by the limitations of existing approaches, our aim is to create a symbolic execution engine that can handle any updates to the internal state of the program. Ideally, our method should also be applied without the need of an input grammar. And lastly, the method should work for any conditions in the program under test, even conditions that have no apparent objective for which to optimize.

4

Execution Model

A program can take on arbitrary forms. It can retrieve input from different sources, jump to arbitrary locations in the code, or be nondeterministic. To reason about the behavior of a program, we define a single execution model. Although this model restricts the types of programs that can be analyzed, we assume the execution model is general enough to allow programs that do not strictly follow this execution model to be transformed into this execution model while being functionally equivalent. For programs that can not be transformed into this execution model, extensions could be made to our work to support their analysis. We have based the execution model on the RERS challenges[15, 12, 11], to allow us to directly apply the model to these problems without needing such a transformation step.

The basis of the execution model is an input-output pattern that mimics a mealy machine. After giving an input to the program, it does an arbitrary computation and then outputs a result. Once it outputs a result, it asks for new input. This loop continues until the program exits. Each input is always one of the symbols of a predefined input alphabet. The program can loop forever when supplied with the right continuous stream of inputs. After the input, the computation can modify the internal variables of the program. Consequently, giving the same input symbol to the program multiple times in a row might lead to different results. However, the internal state resets after restarting. The program will again behave the same and is deterministic.

A multi-path loop is one way to implement such an execution model. This loop retrieves the input and has multiple branches with conditions over the current input symbol and the internal state. Each branch can do a computation and possibly change the internal state. Algorithm 1 shows the pseudocode of the execution model.

Algorithm 1 Execution Model

```
1:  a ← 0                                          ▷ Internal state
2:  b ← 1
3:  c ← 2
4:  loop
5:      i ← input()                                ▷ Retrieve input
6:      if i = 1 then
7:          a ← a + 1                              ▷ Change internal state
8:      end if
9:      if i = 2 ∧ a = 2 then
10:         b ← b + 1                              ▷ Change internal state
11:     end if
12:     ···                                        ▷ More branches that change internal state
13:     if i = −1 then
14:         exit
15:     end if
16: end loop
```

5 Detecting Loop Structures

This chapter focuses on answering our first research question. We repeat the question here:

RQ1: How can loop structures in symbolic execution be detected?

For this question, we will consider methods to detect loops in a program. After detection of a loop and following a path trough that loop, we want to know whether that same path through the loop can be repeated. We chose the term loop structure to incorporate both of these notions.

5.1 Detecting the Loop

In this section, we look at different methods for detecting a loop. Some methods require access to the source code, while others only rely on the machine code or the intermediate representation that gets generated at compile-time.

5.1.1 Source Analysis

One of the simplest ways to get the models from a loop is by analyzing the source code of a program. In the source code of most programming languages, loops are explicitly defined as a `for` or `while` loop. These loops can be extracted from the source code and then used as extra information during symbolic execution.

5.1.2 CFG Analysis

From all program representations, from machine to source code, a CFG can be created. A CFG represents the transitions between different basic blocks in a program, where each basic block contains a sequence of statements. The transitions between the blocks are created from conditional jumps. Where a conditional jump determines which basic block to jump to next based on a condition. The CFG is a directed graph with basic blocks as vertices in the graph. The edges of these directed graphs are the conditional or unconditional jumps in the program. From these directed graphs, loops can be detected. These loops then correspond to loops in the program.

A downside of this approach is that some transitions in the graph can never be executed in practice. Additionally, the basic blocks in the CFG have to be linked back to the running code while doing symbolic execution. To make this linking easier, we will look at a detection method that works during the runtime of the program.

5.1.3 Stack-Based Detection

The traditional runtime model of a program is based on having a stack. On the stack, local variables are stored in stack frames. Upon function calls, a new stack frame is created for that function. The stack frame contains the local variables of that function, the saved instruction pointer, and the saved base pointer. The base pointer is sometimes also called the stack pointer. By saving the instruction and base pointers, execution can be resumed to the function that was running previously. In a loop, the same set of instruction is executed repeatedly. After an iteration of the loop, the program jumps to the start of the loop by setting the instruction pointer to the first basic block of the

loop. To detect this loop behavior at runtime, we can keep track of the instructions that are visited for each stack frame. When we enter a new stack frame, the visited instructions are only tracked in the current stack frame, which prevents two calls to the same method to count as a loop. The most straightforward way to keep track of these instructions is by saving the value of the instruction pointer, since the instruction pointer is unique for each instruction. After each jump instruction, we can check whether this new location has already been visited in the current stack frame.

A limitation for this method is the inability to detect loops resulting from recursion. In recursion, a function calls itself. Every time this function calls itself, a new stack frame is created. This new stack frame has not visited any of the instructions in the function, therefore no loop will be detected. The impact of this limitation can sometimes be disregarded in practice, as some recursive methods in the code will get converted to nonrecursive functions at compilation. This is the case when tail-recursion optimizations can be applied. This optimization exists to reduce the impact of creating a new stack frame for each recursive call.

5.1.4 Execution Model

Due to the assumptions on the execution model, there is another way to detect the main input-output loop. The main input-output loop retrieves input at only one location. Therefore, every time a new input is retrieved, the program is at the same location in the program. To detect a single iteration in the loop, detecting when input is received is sufficient. When a new input is retrieved, one iteration has been made through the loop since the retrieval of the previous input. Since this method is simple yet effective for the RERS problems, it is the method we chose to use.

5.2 Repeatable Path

After detecting a path through the loop, we want to check whether the same path can be followed once more. The intuition behind being able to follow the same path twice is that such a path might be executed an arbitrary number of times. However, there is no guarantee that following a path twice allows repeating it an arbitrary number of times. For a loop path to be repeatable, all the conditions in the path need to be equivalent in the next iteration. If a condition was true on the first iteration through the loop, that condition needs to remain true on the second iteration, if a condition was false on the first iteration, the condition needs to be false on the second iteration. In symbolic execution, a loop path through the program has a constraint that gets generated when following that path. We call this constraint the loop constraint. Just like a path constraint, the loop constraint contains constraints for all conditions as well as constraints for all updates to internal variables. By extending the loop constraint with a slightly updated version of itself, we can simulate the behavior of repeating the same path. The extended loop constraint can be formed from a copy of the loop constraint. In this copy, all symbolic variables that were assigned a new value need to be updated so that it reflects executing the loop path again. If a path through the loop contains the assignment `i = i + 1`, and an if statement with the condition `i > limit`, the loop path that does not trigger this if statement has the following loop constraint: $i_1 = i_0 + 1 \wedge \neg(i_1 > limit)$. This path only updates the `i` variable, so the extended loop constraint can be formed by increasing the subscripts of the `i` variables in the loop constraint by one: $i_2 = i_1 + 1 \wedge \neg(i_2 > limit)$.

If the loop constraint combined with the extended loop constraint is still satisfiable, the same path through loop can be repeated. This is true since the loop constraint contains all the conditions of the path through the loop, and the extended loop constraint represents taking that same path again. So if they are satisfiable in conjunction, the path can be executed at least twice.

5.3 Self Loop

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 all the variables at the end of the loop in the same state as before inputting the symbol, have no effect as well.

To detect these self loops, the path constraint and a special self loop constraint can be given as a formula to the SMT solver. The self loop constraint checks whether there are no changes to the internal state. The self loop constraint can be formed as follows. Let $a, b \cdots z$ denote all variables in the internal state. The symbolic shadow variables representing their respective values before

```
1  int containsEvenNumberOf1s(char* trace) {
2    int j = 0; // Loop variable
3    char symbol; // Character in input
4    int even = 1; // Internal state
5
6    while ((symbol = trace[j++]) != 0) { // Get next character in input
7      if (symbol == '1') {
8          if (even) {
9              even = 0;
10         } else {
11             even = 1;
12         }
13     }
14   }
15   return even;
16 }
```

Figure 5.1: Example function which checks whether there are an even number of '1' characters in the input

executing the loop are $\underline{a}, \underline{b}, \cdots \underline{z}$. With $\overline{a}, \overline{b}, \cdots \overline{z}$ being their symbolic values after the loop (note that for any variable that is not assigned, these are already equivalent). The self loop constraint can be found in the following equation.

$$\underline{a} = \overline{a} \wedge \underline{b} = \overline{b} \wedge \cdots \wedge \underline{z} = \overline{z} \tag{5.1}$$

A self loop exists if the path constraint, including the loop constraint, in conjunction with the self loop constraint is satisfiable.

5.4 Multiple Loop Iterations

For loop paths, we only considered paths that take one iteration through a loop. However, in some cases, the effects of inputting one symbol can be reversed by inputting another symbol. Take for example a function which checks whether there is an even number of '1' characters in a string, the code of such a program is show in Figure 5.1. A single iteration over an input with 'i' does not yield a repeatable path, because the `even` variable is changed and checked in the loop, therefore the same loop path can not be repeated. Instead of only considering loop paths through one iteration of a loop, we can also consider paths that take multiple iterations through the loop. The detection remains the same, but instead of taking the path since the last iteration of the loop, you can take the path through the multiple iterations of the loop. The loop constraint then captures a path over multiple iterations through the input-output loop in the program. For the `containsEvenNumberOf1s` example, the loop constraint over two iterations of inputting a 'i' character is repeatable. In fact, this loop path is a self loop.

5.5 Conclusion

To recap, we have shown different techniques to detect loops. Both the stack-based detection methods and CFG analysis are generally applicable to any program. The stack-based detection method is easy to integrate into symbolic execution as it uses runtime behavior. For the rest of this thesis, however, we will use the simplest solution that works. Based on the execution model, we can use a simple technique that triggers on the input retrieval to detect a path through a loop. A loop path can then be checked whether it is repeatable, if the loop path is repeatable, they might be part of a looping structure. These loop paths can then be used for generalization. Generalization of loop paths is part of the next research question and is discussed in the next chapter.

Loop paths with assignments which do not alter the internal state after a whole iteration are self loops. Since executing self loops has no effect on further execution, these loops are especially interesting for pruning states in symbolic execution. Chapter 7.

6 Loop Generalization

This chapter focuses on our second research question. We repeat the question here:

> **RQ2**: Is it possible to generalize the assignments in loop structures into constraints for a symbolic executor?

6.1 Generalizing Repeated Addition

For programs with loops over a sequence of (unconstrained) symbolic values with only addition, we can construct path constraints that generalize over executing these loops an arbitrary number of times. This section will explain the technique. The general idea relies on the fact that multiple additions are equivalent to multiplication.

6.1.1 Generalization

The next step is to create a generalized constraint over executing the loop an arbitrary number of times. Doing repeated addition is equivalent to multiplication. We introduce a new variable that denotes the number of times the loop is executed. For the example in Table 6.1 this is n. We modify the new constraint to use this variable to form a generalized constraint over the addition in the loop: $\neg(i_1 > limit) \wedge i_2 = i_1 + n \wedge n \geq 0$. There is however an issue, as the variable i can now be updated to any value larger than the limit, we also need to add a constraint that the condition still holds for the final value: $\neg(i_2 > limit)$.

6.1.2 Forming the Input

After forming a generalized loop constraint, we add the constraint to the current path constraint and continue the symbolic execution as normal. Due to the generalized constraint, more paths can be solved. When a new condition is encountered and solved, the model is retrieved from the solver. The model needs to be converted into a concrete sequence of inputs. For all generalized constraints, any constraint on the input needs to be repeated based on the free variable in the model, which is n in the example. If the model states that $n = 10$, the concrete input contains a pattern of 10 repeated symbols.

Name	Constraint
First constraint	$i_0 = 0 \wedge limit = 10$
Second constraint	$i_0 = 0 \wedge limit = 10 \wedge \neg(i_0 > limit) \wedge i_1 = i_0 + 1)$
Loop constraint	$\neg(i_0 > limit) \wedge i_1 = i_0 + 1$
Extended loop constraint	$\neg(i_1 > limit) \wedge i_2 = i_1 + 1$
Generalized loop constraint	$\neg(i_1 > limit) \wedge i_2 = i_1 + n \wedge n \geq 0 \wedge \neg(i_2 > limit)$

Table 6.1: Example of constraints used to detect loops

Assignment	Generalization	Constraints
$i \leftarrow a$	$i \leftarrow a$	a is constant
$i \leftarrow i + a$	$i \leftarrow i + an$	a is constant
$i \leftarrow ai + b$	$i \leftarrow \begin{cases} i + bn, & \text{if } a = 1 \\ ia^n + b\frac{1-a^n}{1-a}, & \text{otherwise} \end{cases}$	a and b are constant
$i \leftarrow (i + a) \bmod b$	$i \leftarrow (i + an) \bmod b$	a and b are constant

Table 6.2: Assignment expressions that are generalizable when repeated. In all the generalization expressions, the variable n indicates the number of times the assignment expression is repeated

6.1.3 Loop Generalization

There are more looping expressions than just incrementing a variable by one. Repeated addition is equivalent to multiplication. Repeated multiplication is exponentiation. Table 6.2 shows a few more expressions that we have generalized. Variables inside a loop path that are not reassigned can be considered constants. If an assignment expression includes such a variable, the variable is interpreted as a constant. Most expressions are straightforward, and we have left out their derivations for brevity. The derivation for repeatedly executing the expression $i \leftarrow ai + b$ is shown in Equation 6.1.

$$
\begin{aligned}
i_1 &= ai_0 + b \\
i_2 &= ai_1 + b = a(ai_0 + b) + b = a^2 i_0 + ab + b \\
i_3 &= ai_2 + b = a(a^2 i + ab + b) + b = a^3 i_0 + a^2 b + ab + b \\
i_4 &= ai_3 + b = a(a^3 i_0 + a^2 b + ab + b) + b = a^4 i_0 + a^3 b + a^2 b + ab + b \\
i_n &= ai_{n-1} + b = a^n i_0 + b(a^{n-1} + a^{n-2} + \ldots + a^0)
\end{aligned}
$$

Using the finite series

$$\sum_{k=0}^{m} z^k = \frac{1 - z^{m+1}}{1 - z} \tag{6.1}$$

$$
\begin{aligned}
i_n &= a^n i_0 + b(a^{n-1} + a^{n-2} + \ldots + a^0) = a^n i_0 + b * \sum_{k=0}^{n-1} a^k \\
&= a^n i_0 + b\frac{1 - a^{n-1+1}}{1 - a} \\
&= a^n i_0 + b\frac{1 - a^n}{1 - a}
\end{aligned}
$$

6.2 Initial Experiments

We implemented the generalized repeated addition for the simple program shown in Figure 1.1 by manually instrumenting the program. We implemented the loop detection, generalization, and then forming the input. When running these experiments, the technique generated inputs that visit the limit in seconds, even for arbitrarily large limits.

This implementation needs manual instrumentation for loops, assignments, and if statements. In practice, this is error-prone to do by hand and should be done automatically. We could have implemented this, however the generalizations for these expressions are not applicable to any loop path. One of our motivating factors was the limitation of current methods to work on any loop. To support any loop, we need a new method to create loop expressions for any loop path.

6.3 Dealing with Non-generalizable Loop Paths

The formulas derived in Table 6.2 provide insight into simple statements. However, it is hard if not impossible to create generalized expressions for arbitrary expressions, especially if multiple assignment statements are present in a loop. A way to deal with this is to add loop constraint multiple times to the model, for each iteration through the loop.

Although the code shown in Figure 1.1 is generalizable using the formulas mentioned above, we will explain the generation of constraints for arbitrary expressions using it. In this example, only one variable is assigned a new value: variable i. For now, we assume the case where just one variable is reassigned, and that only one reassignment is done. We will later expand this to deal with loop paths where multiple variables are reassigned, and where each variable can be reassigned multiple times.

Given the following loop constraint: $\neg(i_0 > limit) \wedge i_1 = i_0 + 1$, and the fact that we assigned a new value to the variable i, we can create the loop path that represents executing the loop one more time. For every assignment, we replace the current variable with a new one. For example: the current variable i_1 is replaced with i_2. Additionally, all the original variables in the path constraint need to be updated with the current assignment, so i_0 will become i_1. Combining both steps: after replacing i_1 with i_2 and i_0 with i_1, we get the loop path for executing this loop again. In the example, this yields the following extended loop constraint $\neg(i_1 > limit) \wedge i_2 = i_1 + 1$. Because of the subscript notation, creating the extended loop constraint from the loop constraint can be done by replacing every i_k with i_{k+1} where k is an integer.

The path constraint after going through the loop n times, the unrolled loop constraint, is shown in Equation 6.2.

$$\begin{aligned} &\neg(i_0 > limit) \wedge i_1 = i_0 + 1 \wedge \\ &\neg(i_1 > limit) \wedge i_2 = i_1 + 1 \wedge \\ &\vdots \\ &\neg(i_n > limit) \wedge i_{n+1} = i_n + 1 \end{aligned} \tag{6.2}$$

6.3.1 Assignments to Multiple Variables

Up to now, we only considered generalizing loops where one variable is reassigned, but the concepts can be extended to loop constraints where more than one variable is reassigned. Forming the extended loop constraint is done similarly as shown earlier. By taking the loop constraint and for every reassigned variable in the loop path, increasing the subscript of all the corresponding variables in the loop constraint by one creates the extended loop constraint.

When there are multiple assignments in the current loop path, it is still possible to form an unrolled loop constraint. Instead of only replacing the subscript for a single variable, all the variables that get a new assignment must be incremented. With these reassigned variables denoted as $a, b, \cdots, z$, and P denoting the loop constraint. The extended loop constraint can be created by substituting every a_i with a_{i+1}, every b_i with b_{i+1} up to substituting every z_i with z_{i+1}. For substitution, we will use the following notation: $[a_i/a_i + 1]P$ for replacing every a_i with a_{i+1} in P, between the square braces, there can be multiple substitutions, and each variable in P is only substituted once. The updated unrolled loop constraint can be seen in Equation 6.3

$$\begin{aligned} &[a_i/a_{i+1}, b_j/b_{j+1}, \cdots, z_k/z_{k+1}]P \wedge \\ &[a_i/a_{i+2}, b_j/b_{j+2}, \cdots, z_k/z_{k+2}]P \wedge \\ &\vdots \\ &[a_i/a_{i+l}, b_j/b_{j+l}, \cdots, z_k/z_{k+l}]P \end{aligned} \tag{6.3}$$

6.3.2 Multiple Assignments to Multiple Variables

Up to now, we have assumed that each variable is reassigned only once in a single loop path. It can also occur that a variable is assigned to multiple times. In these cases, replacing a_i through z_i by a_{i+1} through z_{i+1} respectively in the loop constraint does not represent executing the loop once more and yields incorrect results. Let us take the following loop constraint: $a_1 = a_0 + 1 \wedge a_2 = a_1 * 2$,

where the variable a is first increased by one and then doubled. Increasing all subscripts by one results in the constraint $a_2 = a_1 + 1 \wedge a_3 = a_2 * 2$, but this is not correct, because a_2 is constraint twice when combined with the original loop constraint. The solution is to increase the subscripts by two. Or in general, increase the subscripts by the number of reassignments to that variable. In this case, a is reassigned twice. Once for incrementing it by one and once for doubling it. Therefore, all the subscripts need to be increased by 2. With the example, the extended loop constraint becomes $a_3 = a_2 + 1 \wedge a_4 = a_3 * 2$. To generalize this to multiple variables with multiple assignments, we need an additional symbol for each variable, representing how many times an assignment is done in the loop path. For this, we use the bar above the corresponding variable name, so $\overline{a}$ represents the number of times the variable a is reassigned in the loop path. In Equation 6.4 the revised unrolled loop constraint is denoted.

$$\begin{aligned} &[a_i/a_{i+1*\overline{a}}, b_j/b_{j+1*\overline{b}}, \cdots, z_k/z_{k+1*\overline{z}}]P \wedge \\ &[a_i/a_{i+2*\overline{a}}, b_j/b_{j+2*\overline{b}}, \cdots, z_k/z_{k+2*\overline{z}}]P \wedge \\ &\vdots \\ &[a_i/a_{i+l*\overline{a}}, b_j/b_{j+l*\overline{b}}, \cdots, z_k/z_{k+l*\overline{z}}]P \end{aligned} \tag{6.4}$$

6.4 Conclusion

In this chapter, we first looked at generalizing loops with simple update statements. We provided generalizations for these update statements to capture the effects of these updates statements on internal variables in the program. These generalizations allow one to calculate the values of the internal variables after running these update statements multiple times. It is non-trivial to create generalization expressions for any update statements. To still use the effects of repeating a loop path with any update statements, we create the unrolled loop constraint. This loop constraint captures the changes of internal variables when executing the loop an arbitrary number of times.

7

Loop Structures in Symbolic Execution

This chapter defines our method to use looping structures in symbolic execution. In this chapter, we focus on the third research question:

RQ3: How can loop structures be used to reach new branches in symbolic execution?

The goal of this chapter is to use the unrolled loop constraint from the previous chapter to create a path constraint that represents taking an arbitrary number of iterations through the loop over the detected loop path.

7.1 Iteration Constraint

Although a generalized constraint is not possible, we can create a disjunction of executing the loop once, executing the loop twice or executing the loop thrice, and so on.

The final value of the variable i after executing the loop an arbitrary number of times is denoted by i_f. This results in the following disjunction, $(i_1 = i_f \wedge j = 1) \vee (i_2 = i_f \wedge j = 2) \vee (i_3 = i_f \wedge j = 3) \vee \cdots \vee (i_l = i_f \wedge j = l)$ where l is the maximum number of times we can execute the loop using this constraint. We call this disjunction the iteration constraint. Notice that we added additional constraints over the variable j, their purpose will become apparent later and can be ignored for now.

What value to choose for l is still a decision to be made, and for now we left this as a choice to the user. Choosing a higher number of l leads to a larger constraint, but represents executing the loop more times. Having a larger constraint can lead to an increased runtime for the solver.

During symbolic execution, we can use the unrolled loop constraint and the iteration constraint to create a path constraint representing executing the loop an arbitrary times up to the limit.

After forming both the unrolled loop constraint and the iteration constraint, they are added to the current path constraint, and symbolic execution is continued. As usual, upon encountering a branch, the solver is called with the current path constraint and the (possibly inverted) branch condition. If this yields satisfiable, an input needs to be constructed from the model. To create the input, we need to know the number of times the loop needs to be executed. For this purpose, we have added the additional constraint on j to every disjunction in the iteration constraint. The value of j denotes the number of times the loop is required to execute to reach the given path. Upon getting a model that satisfied the path constraint, we can inspect the value of j to only append the symbols to the new input trace that are inside the loop when executing it j times.

To minimize the length of the input, an optimization strategy can be added to the solver. With an optimization strategy to minimize j, the solver still satisfies the path constraint while also minimizing the value of j. This makes sure that the loop is executed the least the number of times necessary while still reaching the path it is solving.

7.1.1 Assignments to Multiple Variables

For the iteration constraint, we assumed there was only one variable that was reassigned. However, multiple variables can be assigned to in a loop path. In the previous chapter on generalizations, we

make the same extension to assignments to multiple variables.

For the iteration constraint, instead of having just one variable taking one of the values in the loop, all the variables need to be equivalent to the same assignment in the loop. Equation 7.1 shows the revised iteration constraint. The variables a through z are again the reassigned variables. a_f through z_f represent the final values of a through z after doing an arbitrary number of (but at most l) iterations through the loop. Just like earlier, l represents the number of iterations through the loop, a configurable parameter.

$$\begin{aligned}(j = 1 \wedge a_1 = a_f \wedge b_1 = b_f \wedge \cdots \wedge z_1 = z_f) \vee \\ (j = 2 \wedge a_2 = a_f \wedge b_2 = b_f \wedge \cdots \wedge z_2 = z_f) \vee \\ \vdots \\ (j = l \wedge a_l = a_f \wedge b_l = b_f \wedge \cdots \wedge z_l = z_f)\end{aligned} \tag{7.1}$$

Note the placement of the $\vee$ and $\wedge$ operators. You could also suggest creating a loop constraint where each variable needs to equal one of the values in the loop. This is incorrect, as one variable could then take on the value of executing the loop once, where another value could take on the value of executing the loop twice. All variables need to be equivalent to the same iteration.

7.1.2 Multiple Assignments to Multiple Variables

Just like we mentioned when creating the unrolled loop constraint in Section 6.3.2, a loop path can include multiple assignments to the same variable.

The iteration constraint needs updating as well, as the variables must only be equivalent to one of their values after doing a whole cycle in the loop, not to any of its intermediate values during the loop. For this, we again need the number of assignments for each of the variables in the loop path. Equation 7.2 shows the updated iteration constraint.

$$\begin{aligned}(j = 1 \wedge a_{1+0*\overline{a}} = a_f \wedge b_{1+0*\overline{b}} = b_f \wedge \cdots \wedge z_{1+0*\overline{z}} = z_f) \vee \\ (j = 2 \wedge a_{1+1*\overline{a}} = a_f \wedge b_{1+1*\overline{b}} = b_f \wedge \cdots \wedge z_{1+1*\overline{z}} = z_f) \vee \\ \vdots \\ (j = l \wedge a_{1+(l-1)*\overline{a}} = a_f \wedge b_{1+(l-1)*\overline{b}} = b_f \wedge \cdots \wedge z_{1+(l-1)*\overline{z}} = z_f) \vee\end{aligned} \tag{7.2}$$

With the updated iteration constraint and the unrolled loop constraint added to the current path constraint, symbolic execution can continue to execute. Whenever a variable is used that was updated in the unrolled loop constraint, the symbolic value of that variable must match the final value in the iteration constraint. In the construction in Equation 7.2, we used a_f through z_f for variables a through z respectively for the symbolic values after executing the loop path. After adding these constraints to the path constraint, the path constraint now represents taking 1 up to l iterations of that same loop path. It can be seen as executing all these iterations at the same time. When solving for new branches, the solver will find an iteration that satisfied that branch. If this branch cannot be satisfied, this branch cannot be reached with the current path constraint.

7.2 Conclusion

In this chapter, we introduced the concept of the iteration constraint which ensures that the values of all internal variables are equal to one of the iterations through the loop. The iteration constraint can be created from a loop path. Combined with the unrolled loop constraint created from that same loop path, a symbolic executor can create a path constraint that represents following a loop path once, twice, all the way up to the unrolled loop amount.

8

Symbolic Execution Experiments

This chapter contains the experiments and results of applying the concepts outlined in Chapter 7 in a custom symbolic executor. These results allow us to answer the fourth research question:

> **RQ4**: Does detecting and using loop structures allow symbolic execution to cover more states of a program?

8.1 Extension to Klee

To allow comparing our method for loop detection to the traditional approach, we wanted to implement our loop detection as an extension to Klee. This would result in an equivalent execution speed for both of the methods. When inspecting the Klee source code, we had troubles to understand its behavior as the code is heavily optimized. After a long time of struggling to find out how to see the symbolic constraints in Klee, we were able to print them out. Our goal was to use these constraints to reconstruct a loop path and check whether this path was repeatable according to Section 5.2. However, the only constraints we were able to get were a few constraints over the free input variables. We could not find constraints for any of the internal variables, or other conditions in a simple program. When taking a step back, we realized this was the expected behavior of Klee: it optimized all the other expressions, since all the variables had concrete values. This was a major hurdle, and we tried to disable these optimizations. In the end, we were able to disable some of them, but not all. Unfortunately, we had to cut our losses and give up on making an extension to Klee. We could have persistent and probably figured this out, but this would require much more time than planned. Instead, we focussed on testing the methods themselves.

8.2 LLVM IR Symbolic Executor

As a second attempt, we began implementing a symbolic execution engine from scratch based on the LLVM Immediate Representation. Since many programming languages are compiled to LLVM IR, and Klee also uses this, we wanted to build a symbolic execution engine based on this instruction set. Although the instruction set for LLVM is relatively small, this still was not a small task. We gave ourselves two weeks to see how far we would get. If after these two weeks, we made significant progress towards a symbolic execution engine, we would continue this path. After two weeks, we had a lot of functionality, however, we again discovered a major hurdle: pointer casting. We did not foresee pointer casts to exist in the programs we were testing with, as the programs in C did not use pointer casts. Unfortunately, initialization of arrays makes heavy use of pointer casts. Using pointer casts is one thing, but having to implement them correctly is a completely different story. We again had to drop our plan and switch to plan C: create a symbolic executor based on simple instrumentation and a limited instruction support. This led us to the creation of SymLoop . The rest of the chapter explains our implementation of our method in this Java-based symbolic executor.

8.3 Building SymLoop

We built a symbolic executor with the ability to detect and create constraints for loops over its input using the non-generalizable formulas from Section 6.3. This allows us to test our methodology. Our goal was to extend the existing symbolic executor Klee to be able to compare apples to apples.

8.3.1 SymLoop

The symbolic executor is implemented in Java and can only run symbolic execution on Java programs with strict requirements. The program-under-test follows the execution model outlined in Chapter 4. The Java versions of the RERS problems of 2020 follow this execution model. Figure 8.1 shows an example of the minimal structure. If the input reaches an error state, the custom error verifier is called.

Our symbolic executor, from here on referred to as 'SymLoop', first instruments the code to be able to form constraints for the SMT solver. During the instrumentation phase, every variable gets a separate shadow variable that corresponds to this variable in the SMT solvers' representation. Any changes to the original variables are reflected in their corresponding shadow variable. The programs are instrumented to retrieve input from the symbolic executor. The output of the program is also passed to the symbolic executor. On every conditional branch, the symbolic executor gets invoked with the branch condition. This allows the symbolic executor to generate inputs that execute different paths through the program. This instrumented version then gets compiled and can be run with SymLoop linked as a library. This process starts the symbolic execution. The SMT solver Z3 [18] is used for all the satisfiability queries. To compare the results of SymLoop versus standard symbolic execution, we can disable its loop detection entirely, leaving a standard symbolic executor with the same performance characteristics of SymLoop.

Due to our assumption of the execution model, we know that every time a new input is retrieved, the program is in the same location in the loop. Right before inputting a new symbol, SymLoop generates the loop constraint since inputting the previous symbol. From the loop constraint the extended loop constraint is created following the steps in the previous sections. If the current path constraint in conjunction with the extended loop constraint is satisfiable, a loop is detected.

Inputting the last symbol again will lead to executing the same path through the program. From this intuition, both the iteration constraint and the unrolled loop constraint are generated and added to the current path constraint. Additionally, SymLoop adds an optimization strategy to the SMT solver to minimize the number of iterations through the loop. The upper limit of the number of iterations through the loop can be specified as an argument for running the symbolic execution. The maximum amount of iterations through the loop we refer to as the unrolled loop amount and in the results, it is denoted with the letter `l`. If none of the loop paths are satisfiable (so none can be repeated), nothing is added to the path constraint.

Detection depth

Apart from the unrolled loop amount, SymLoop allows users to set the detection depth `d`. The detection depth denotes up to how many symbols back the loop constraint and extended loop constraint are created and checked for satisfiability. For example, if the detection depth is 3, first the loop constraint and extended loop constraint are generated and checked for the path through the program since inputting the last symbol, then the loop path is checked since inputting the second to last symbol, and lastly, the loop path is checked from inputting the last 3 symbols. For the first one that yields satisfiable, the unrolled loop constraint gets added to the path constraint. This functionality allows detecting loops that require up to `d` input symbols in a row.

8.4 Setup of Running SymLoop

To test SymLoop we ran some experiments. We ran SymLoop on the Java versions of the RERS reachability problems of 2020. All the problems contain a multitude of intentional errors. For each problem, we kept track of the time it took SymLoop to find each error. Additionally, we ran Klee [3] on the same problems to be able to compare our solution to the state-of-the-art. For Klee, the same measurements are used: the time it took to find each error.

We chose to run SymLoop with two different sets of parameters. One version ran with a detection depth of 5 and an unrolled loop amount of 10. The second version ran with 10 and 50 for the

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ProblemPowersOfTwo {
    static BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

    private String[] inputs = { "i", "p"};

    public int i = 0;
    public boolean cf = true;
    public boolean done = false;

    public void calculateOutput(String input) {
        cf = true;
        if (cf && input.equals("i")) {
            i += 1;
            cf = false;
        }
        if (cf && input.equals("p")) {
            cf = false;
            if (i > 128) { Errors.__VERIFIER_error(7); }
            if (i > 64) { Errors.__VERIFIER_error(6); }
            if (i > 32) { Errors.__VERIFIER_error(5); }
            if (i > 16) { Errors.__VERIFIER_error(4); }
            if (i > 8) { Errors.__VERIFIER_error(3); }
            if (i > 4) { Errors.__VERIFIER_error(2); }
            if (i > 2) { Errors.__VERIFIER_error(1); }
        }
    }

    public static void main(String[] args) throws Exception {
        // init system and input reader
        ProblemPowersOfTwo eca = new ProblemPowersOfTwo();

        // main i/o-loop
        while (true) {
            // read input
            String input = stdin.readLine();

            try {
                // operate eca engine output =
                eca.calculateOutput(input);
            } catch (IllegalArgumentException e) {
                System.err.println("Invalid input: " + e.getMessage());
            }
        }
    }
}
```

Figure 8.1: Example of the Java structure necessary to run our custom symbolic executor

detection depth and unrolled loop amount respectively. The choice of these parameters is somewhat arbitrary, but the intuition was to have one for detecting smaller loops that had a bit less overhead and one that was able to detect bigger loops but probably would be slower due to the additional overhead of doing longer loops over more inputs.

Along with the two versions of SymLoop with the loop detection enabled, we ran one version that had loop detection fully disabled. This version we consider to be the baseline of most of our testing. We could have opted for Klee to be the baseline, but due to its many optimizations, it runs significantly faster and does not allow us to solely compare the benefits of adding the loop detection.

8.4.1 Results of Running SymLoop

The initial runs of SymLoop on the RERS reachability challenges of 2020 yielded results that indicated the solution was significantly worse compared to the baseline (remember, SymLoop with loop detection disabled). The experiments involved running the tool, so either Klee or SymLoop , for 10 minutes on each problem.

As we expected, SymLoop was no match for Klee, as Klee is heavily optimized. Still, it provides insight into how many of the errors a state-of-the-art tool could find in the specified time budget. All errors found by SymLoop were also found by Klee, and almost always within a fraction of the time.

SymLoop with loop detection enabled, failed to find many errors on a handful of problems within the 10-minute time budget. On other problems, it failed to find some errors. When comparing SymLoop with the baseline, both versions find fewer errors. On Problem 12, SymLoop is not able to find any errors. And for every problem, the baseline finds the same or more errors than SymLoop finds. The main takeaway of the first run was that the loop detection in SymLoop was performing way worse when compared to the baseline. We expected there to be some overhead to loop detection, but we did not expect such a large discrepancy.

After further investigation and profiling the loop detection, we found some culprits to the slowdowns that allow improving SymLoop .

8.5 Improvements

Profiling SymLoop indicated that a significant portion of the time was spent on generating the unrolled loop constraint by substituting each variable in the constraint individually. The SMT library Z3 has a method to substitute multiple variables in a constraint at once, for which it only has to traverse the whole constraint once instead of once for each variable. Making use of this functionality was enough to mitigate this issue while being functionally equivalent.

Profiling indicated that the runtime of SymLoop is dominated by calls to the SMT solver Z3. So the goal of the next improvements is to reduce the number of calls to the solver.

Another issue with creating the unrolled loop constraint was that after adding each iteration of the unrolled loop constraint, SymLoop would give the current constraint to the solver to check if it was still satisfiable. The goal of this was to stop on loops that were not repeatable of at least the unrolled loop amount. However, in most cases, the constraint was still satisfiable, as it was a repeatable loop. We adjusted the strategy such that instead of checking after each iteration of the unrolled loop constraint, SymLoop only checks once at the end. If the original path constraint in conjunction with the unrolled loop constraint is not satisfiable, the path is not considered a loop, and the symbolic execution continues as normal. Again, this change was functionally equivalent to the original behavior, but skips a lot of unnecessary calls to the solver.

A method that requires a bit more explanation is the optimization to skip solving for branches as long as the path constraint (including any loop paths) does already capture the current input. Let us illustrate this with an example: if a program has a looping path over the input `i`, after seeing this path once, the loop is detected and the path constraint is extended with an iteration constraint and unrolled loop constraint for this looping path. This path constraint already captures the path constraint for inputs `ii`, `iii`, `iiii`, etc. If the next symbol in the input is actually an `i`, the full input up to now would be `ii`. Since the path constraint already captures having run this input, solving any encountered branch only has to be done once. If the next symbol is again an `i`, no additional calls to the solver have to be made. This can be skipped as long as the input matches

the looping pattern. Note that the path constraint does not represent going through the loop an infinite amount of the pattern, but up to the unrolled loop amount, from that point solving needs to be enabled again. If the loop continues, after one iteration the loop is detected again and solving can again be skipped.

8.5.1 Results of Running Improved SymLoop

Running the experiments again with the optimized version of SymLoop yielded improved results. Compared to the baseline, it finds most errors a bit slower. However, the two runs of SymLoop are not able to find all the errors within the 10-minute time limit. The version of SymLoop with loop detection enabled performed in most cases only slightly worse than the baseline with loop detection disabled. However, in a few cases, SymLoop was able to find extra errors, for example on problem 13. There, the d10-l50 variant was able to find errors 14 and 65. The following two traces were hitting errors 14 and 65.

```
Error 14: CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGI
     HGIHGIHGIHGIHGIHGIHGIHGIHGIHGIB
Error 65: CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGI
     HGIHGDG
```

A large portion of these inputs repeats and is generated by using an unrolled loop constraint. This indicates our method works as expected and can find some errors that Klee is not able to find in the same time budget.

Both of these errors were found by the same version of SymLoop , the d10-l50 variant, which can exercise longer loops. The d5-l10 variant did not find these errors. When counting the number of repetitions in the inputs, the first one is 37 repetitions of `IHG`, and the second error is 29 repetitions of the same pattern. This explains why the d5-l10 variant is unable to capture the errors, as the unrolled loop amount is 10. Given enough time, it would theoretically be able to find them, but in practice, this would already require a sufficiently long trace.

8.6 Increasing the Time Limit

Since Klee found different errors, we expected the time budget of 10 minutes to be too limiting for SymLoop . To investigate whether giving it more time would also allow SymLoop to find the same errors that Klee found, we increased the time budget to 2 hours. This still allows us to run all the problems sequentially and be done within a day. Alongside the 2-hour runs of the two versions of SymLoop , we reran Klee and SymLoop with loop detection disabled for the same 2 hours on each problem to make a fair comparison.

Klee is not able to find errors 14 and 65 in problem 13. Even running Klee for 24 hours did not result in finding these errors. On problems 12, 13, and 15, SymLoop can find errors that neither Klee nor the baseline was able to find. See the listing below for all the errors and the traces needed to reach the errors. None of these errors were found by Klee or the baseline within 2h. In all of these errors, a repeating pattern occurs, which indicates the usefulness of SymLoop .

```
Problem 12:
 - Error 68: EGIIGAJEIDEGAJEIDEGAJEIDEGAJEIDEGAJEIDEGAJEIDEGJ

Problem 13:
 - Error 14: CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGI
     HGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIB
 - Error 47: CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGEJ
 - Error 65: CIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGIHGI
     HGIHGIHGDG

Problem 15:
 - Error 17: IBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBGJBJBG
     JBJBGJBJBGJE
 - Error 30: LGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDH
     CGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCGCDHCA
 - Error 39: LOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILF
     COILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCOILFCL
 - Error 53: LHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJBHABJ
     BHABJBHABJBHABJBM
 - Error 63: IBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKBCKAKM
```

8.7 Limitations

We have also profiled the runtime of the SymLoop to investigate where the tool spends its time. The majority of its time is spent in Z3 on solving the path constraints. For symbolic execution, spending the majority of the execution time on solving the path constraints is expected. However, the path constraints we generate for loops are larger and contain more possible options for the solver to consider. This reduces the overall speed in terms of executions per second.

Our solution is not perfect for this use case, as Klee was still able to find some errors that SymLoop was not able to find in the same amount of time. For SymLoop to be able to reach these errors will most likely require some of the optimizations that Klee uses to minimize the overhead introduced by the solver. We have implemented the optimizations. In the next section, we discuss some of these optimizations.

8.8 Optimizations

Klee is still able to find most errors in significantly less time compared to SymLoop . As mentioned earlier, we assume most of this is due to the optimizations in Klee. To investigate whether these optimizations improve the runtime, we implemented some of the optimizations in SymLoop .

The most notable optimization is automatic inference of concrete values. When an assignment occurs in symbolic execution, the new value is an expression. In some cases, these expressions only contain concrete values. If the expression only contains concrete values, the new value can be calculated by evaluating the expression. Instead of using the expressions for the variables, we can already evaluate these expressions and give these to the solver. These assignments are simpler than the expressions, and thus allow the solver to more quickly decide on satisfiability.

Additionally, after adding a constraint which makes the model unsatisfiable, adding more constraints cannot make the model satisfiable again. This allows skipping the solver altogether when the model is unsatisfiable. This can be the case when adding a constraint which evaluates to false.

Alongside the concretization of the assignment values, we also scan any added constraints for equality operators. If those equality operators are at the root of the expressions, or there is an and operator at the root with an equality child node, this can be treated similarly to the assignment of variables. If the equality contains a variable on the right or left side and a concrete value on the other side, the variable must be equal to that value.

8.9 Conclusion

In this chapter, we created the symbolic execution engine SymLoop, which uses the techniques described in earlier chapters. The symbolic executor runs on the Java versions of the RERS problems by instrumenting the problems. Any solver queries are handled by Z3. We tested initial versions of SymLoop and were able to find errors that Klee did not find. Howerver, SymLoop suffered from performance problems, so we implemented some optimizations to decrease the solve times.

9 Learning State Machines

In this chapter, we show methods for more efficiently handling loops during equivalence checking in active learning. Our goal is to answer the following research question:

RQ5: How can we use symbolic execution for equivalence oracles in active learning?

In active learning, equivalence checks are used to in the MAT framework to find inconsistencies between a hypothesis model and the system under test.

In the introduction in Figure 1.4, we showed two example models learned from the same program. In one of these models, there is only one state with self loops. In the other model, there is a change detected after executing the 'i' loop for five iterations. In traditional equivalence checks, there is no distinction made between loops and other transitions. To be able to detect the change using the traditional W-method would require a w of at least 5, leading to 2^5 possible paths to verify. Just like the path explosion problem in symbolic execution, the number of paths also grows exponential in relation to the number of states which behave equivalently.

If we have insight into the system under test and expect that there is a looping behavior which changes after a number of iterations, we can more efficiently find counterexamples for a hypothesis model. In these cases, not all paths of a certain length have to be explored, but just the cycles. Our first methodology for detecting these changes uses the same notions as the W-method. However, instead of verifying all paths, it only checks cycles in the hypothesis model.

9.1 Loop-W-Method

The W-method is not able to efficiently find counterexamples for systems that require multiple iterations through a loop before showing different behavior. To illustrate, let d be the number of inputs that are in the loop, and let n be the number of iterations through the loop before it shows different behavior for one of the distinguishing traces. Let Σ denote the input alphabet, a set containing all the input symbols, and let Q denote the states in the hypothesis. Note that $|Q|$ then denotes the number of states as well as the number of access sequences and the number of distinguishing traces. The W-method requires a depth of at least $d \cdot n$. To find a counterexample for such a loop then requires $|Q| \cdot |\Sigma|^{d \cdot n \cdot |Q|}$ membership queries, with the average at half that amount.

To reduce the amount of queries necessary for equivalence checking for loops, we introduce the Loop-W-method. This method requires an additional parameter l. The parameter l denotes up to which depth, the number of iterations, a looping is checked. For each cycle in the hypothesis model and for each distinguishing sequence, the access sequence: A, the loop sequence: L, and the distinguishing sequence D are concatenated. This forms the sequence $A \cdot L \cdot D$. This sequence needs to be formed for all the iterations through the loop up to depth l: $A \cdot L^i \cdot w \cdot D \; \forall \; i \in [1..l]$. If checking any of these sequences results in an output that does not match the hypothesis, it is returned as a counterexample.

9.1.1 Complexity

The Loop-W-method is able to find counterexamples for loops more efficiently. To compare complexity, we need the number of cycles in a system. We use the c to denote the number of cycles in the hypothesis model. To find a counterexample for a looping model requires $c \cdot l \cdot |Q|$ membership queries, since each cycle needs to be checked. In the complexity analysis for the W-method, we assumed the number of iterations through the loop before it shows different behavior to be known beforehand. The configurable parameter l must be at least the number of iterations, therefore we assume l to be configured as n. To find a counterexample for such a loop using the Loop-W-method requires $c \cdot n \cdot |Q|$ membership queries. To better compare this to the complexity of the W-method, we need some bounds on the number of cycles (c) in a hypothesis model.

To get a lower bound for the number of cycles, the hypothesis model must be able to infinitely handle all the input symbols. This can be achieved by having one sink state which has transitions for every symbol in the input alphabet loop back to itself. All other states can be constructed to lead to this final sink state, with no additional cycles. The minimum number of cycles is then defined by the size of the input alphabet $|\Sigma|$.

For a complete hypothesis model, every state must have a transition for every symbol of the input alphabet. Therefore, there are a total of $|\Sigma|^s$ edges. A cycle is a sequence of these edges with a maximum length of s. If the sequence becomes longer, at least one state must be visited twice. If a state is visited more than once, the path is no longer a cycle. A cycle starts in a specific state, so there are s possible starting points. There are only $|\Sigma|$ edges leading out of this state that are possible as subsequent edges. This reduces the maximum number of cycles in a model to $|Q| \cdot |\Sigma|^{|Q|}$. With this upper bound included in the complexity formula, the upper bound for the number of membership queries is $|Q| \cdot |\Sigma|^{|Q|} \cdot n \cdot |\Sigma|$.

For the W-method, there are $|Q| \cdot |\Sigma|^{d \cdot n \cdot |Q|}$ membership queries needed. The W-method is exponential for the number of states, the number of symbols in the loop and the number of iterations through the loop. Our Loop-W-Method is only exponential relative to the number states. This stems from the upper bound on the number of cycles in a model. For models with fewer loops, the Loop-W-method requires even less membership queries.

9.2 Symbolic Execution for Testing Loop Equivalence

In previous section, we have shown the naive method for checking loops in hypothesis models. However, with symbolic execution, we can verify these loops as well. Although the Loop-W-method is applicable in any scenario, the symbolic method is only possible when the underlying behavior of the software is available. The system is no longer a black box, and the symbolic method needs full access to the system, making it a white box method.

Checking for self loops is done by generating the access trace to one of the states in the loop and the subsequent symbols to execute this loop. The access sequence and loop symbols are then run by the symbolic executor. The symbolic executor collects the path constraint from the looping part. This loop path can then be checked for a self loop, as described in Section 5.3. If the loop path is a self loop, the model is correct for this loop and no counterexample can be found.

When the loop path is not a self loop, the unrolled loop and iteration constraints are created to depth l according to Section 6.3 and Section 7.1, respectively. These constraints are added to the current path constraint. Afterward, 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 in to 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

[1]Difference in path does not necessarily mean different behavior. You can construct two paths with equivalent behavior. However, in our experiments this never occurred.

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 are indistinguishable up to depth l. For each tested loop, this check calls the symbolic executor once for each distinguishing trace.

Initially, we experimented with checking all distinguishing traces for a loop in one call to the solver. This resulted in larger models with more constraints. Oftentimes, these models were too complex for the solver to handle, leading to unnecessary long solve times or even timeouts of the solver. When the path through the loop is simpler, and the number of distinguishing traces is minimal, using this methodology might yield better results.

When comparing the symbolic method to the Loop-W-method, the symbolic method uses significantly less membership queries. By collecting the path constraint, checking a loop in a system only requires running the system for each distinguishing trace. When comparing this to the Loop-W-method, the system needs to be run for each iteration in the loop for each distinguishing trace. Additionally, the symbolic method detects changes in the whole path of each distinguishing trace, whereas the Loop-W-method only can detect changes in the last output of the system. The benefit of the Loop-W-method is that it requires no access to the underlying system, making it applicable in any situation, whereas the symbolic method needs to collect the path constraints.

9.3 Experiment Setup

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, the LTL problems of 2020 are too simple and can be learned with W-method equivalence checkers with a shallow depth. This thesis focuses on the reachability problems that require more sophisticated methods to find counterexamples of hypothesis models. We use problems 12, 13 and 15 for active learning.

We implemented the Loop-W-method as an equivalence checker in the LearnLib framework [17, 21]. When the symbolic 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 cycles. After finding the loops, each loop is checked by concatenating the access sequence A with the loop sequence L and one of the distinguishing sequence 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 9.2. SymLoop collects the path constraint, and checks for self loops using the self loop constraint. If it finds a self loop, the symbols that excites this behavior must be self-loops in a learned model. If a loop is not a self loop, it 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. Section 10.3 shows the results.

9.4 Conclusion

Current methods for equivalence checking do not handle loops differently than other transitions. In some systems, these loops can show different behavior. We propose loop equivalence checkers for checking the behavior of the system after many iterations through loops. In black box scenarios, the Loop-W-method can be applied. For systems where path constraints can be collected, the SymLoop equivalence checker can verify the behavior and guarantee equivalent behavior for some self-loops. Unrolling the path constraint allows detecting changes for distinguishing traces when following the loop once in comparison with following the loop multiple times. For systems where running membership queries is costly, symbolic execution reduces the cost associated with model checking. Additionally, symbolic execution can detect changes in paths through the system, whereas the Loop-W-method can only detect changes in the output.

10 Results

This chapter shows the results of running symbolic execution and active learning on the RERS problems of 2020. But first, we look at the results of running symbolic execution on the program outlined in the introduction.

10.1 Running the Initial Problem

In Chapter 1, we outlined the problem of running symbolic execution on a simple program. We have rerun the experiments for this program using SymLoop and have included the results in Figure 10.1. We have run two different versions of SymLoop to illustrate the differences. SymLoop with a loop unroll amount (l) of 0 only detects self loops and skips executing those. SymLoop with a loop unroll amount of 50 also generates constraints for executing any non-self loop up to 50 iterations.

We compare our two versions of our symbolic executor to the state-of-the-art tools when running the same program. Figure 10.1 shows the time to find an error that requires iterations through a loop. Both versions of SymLoop yield much better results compared to Klee and JDart on the same problem, by finding the error significantly faster. Additionally, when JDart did not find an input to cover the code that requires only 16 iterations, SymLoop was able to continue finding errors. There is still a non-linear trend when nearing 200 iterations through the loop to find the error. Increasing the loop unroll can partially solve this issue, but checking for satisfiability takes longer for larger models.

When looking at the log-scaled plot, there is also a small difference in startup time. The startup time of SymLoop is lower than the startup time of Klee or JDart. Since JDart and Klee can run on arbitrary programs, including programs that access the system, this requires more setup time. However, this difference is relatively small, at most 1 second. Therefore, we deem this negligible when the run time of these tools goes up to 400 to 600 seconds.

10.2 RERS Challenges

In Figure 10.2, we have collected the results of running symbolic execution on a few of the RERS reachability problems of 2020. We ran each method for 2 hours and recorded the time it took to find each error. The figure shows the number of errors each method has found over time. For the baseline, we used our original implementation of the symbolic execution engine, with any loop detection or loop unrolling disabled. When creating SymLoop , we also optimized the calls to the solver. These optimizations can be done for the baseline as well, so we also ran the optimized baseline.

For SymLoop , we included two runs, both of the runs use a detection depth of 10 and a loop unroll amount of 50. The run labeled 'SymLoop - intermediate' was our initial version of SymLoop with loop detection enabled, whereas 'SymLoop - optimized' includes all optimizations explained in Section 8.8. The full results are available in Appendix A. To compare to state-of-the-art methods, we have included Klee as state-of-the-art symbolic executor. AFL++ is included as the state-of-the-art fuzzing tool. We have summarized all the results for symbolic execution into Table 10.1. For each different tool or version of SymLoop , the number of discovered errors is shown.

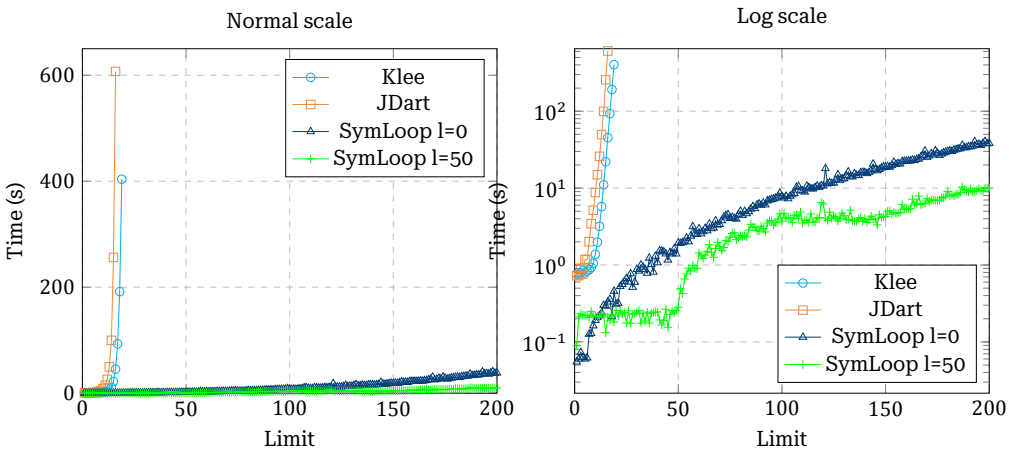


Figure 10.1: Time to run symbolic execution on the example show in Chapter 1. The graph shows the time needed to find an error for a certain limit. For Klee we stopped running past a limit of 19. When running JDart past a limit of 16, it crashed and did not find the error.

Program	d	l	m	P11	P12	P13	P14	P15	P17	P18
Total Errors				18	17	43	15	55	30	42
AFL++			120m	**18**	15	27	**15**	0	**30**	**30**
Baseline	0		10m	**18**	14	22	**15**	36	**30**	1
Baseline - long	0		120m	**18**	15	22	**15**	40	**30**	**30**
Baseline - optimized	0		120m	**18**	16	23	**15**	40	**30**	**30**
Klee			120m	**18**	16	25	**15**	41	**30**	**30**
SymLoop - initial	5	10	10m	**18**	13	0	**15**	7	24	0
SymLoop - initial	10	50	10m	**18**	12	0	**15**	4	24	0
SymLoop - intermediate	10	50	120m	**18**	**17**	26	**15**	**45**	**30**	14
SymLoop - optimized	10	50	10m	**18**	**17**	26	**15**	44	**30**	25
SymLoop - optimized	10	50	120m	**18**	**17**	**35**	**15**	**45**	**30**	**30**

Table 10.1: The number of errors found by each program for each RERS problem. The column 'd' shows the detection depth, and 'l' shows the loop unroll amount. The 'm' column denotes the runtime in minutes for each tool.

10.2.1 Analysis of Results

When looking at the results of Problem 11 and Problem 14 in Figure 10.2, all methods were able to reach the same number of errors in a very short time span. For problems which do not have these behaviors, our results from RERS Problem 11 show that the loop detection has minimal impact on the overall run-time, the total time to find all 18 errors is 1.2 seconds for SymLoop and 1.7 for the baseline. For Problem 14, all 15 errors were found within 12 and 9.4 seconds for the baseline and SymLoop respectively. This shows that for some programs, there is a performance penalty for loop detection. This performance penalty is the result of invoking the solver more often. The solver is invoked more often to check for repeatable loop paths. Additionally, when loops are detected, the unrolled loop constraints and iteration constraints add additional complexity to the path constraint. This extra complexity leads to longer solve times.

Because the number of errors detected on problems 11 and 14 is the same, we will not look further into these results. On Problem 18, all tools except the intermediate version of SymLoop were able to find 30 errors. The intermediate version spent a lot of time on solving constraints, without making much progress. This also indicates the benefit of the optimizations we implemented. We got the most interesting results on Problems 12, 13 and 15. On these problems, SymLoop was able to find more errors within the two-hour time limit. Additionally, these problems show the benefit of optimizing the solver expressions, the optimized version is able to find the same errors on Problem 12 and 15, yet it requires significantly less time. On Problem 13, the non-optimized version is not able to find the same number of errors. Looking more closely at the specific errors that each method was able to find, we see that for some problems, Klee, and the baseline were able to find

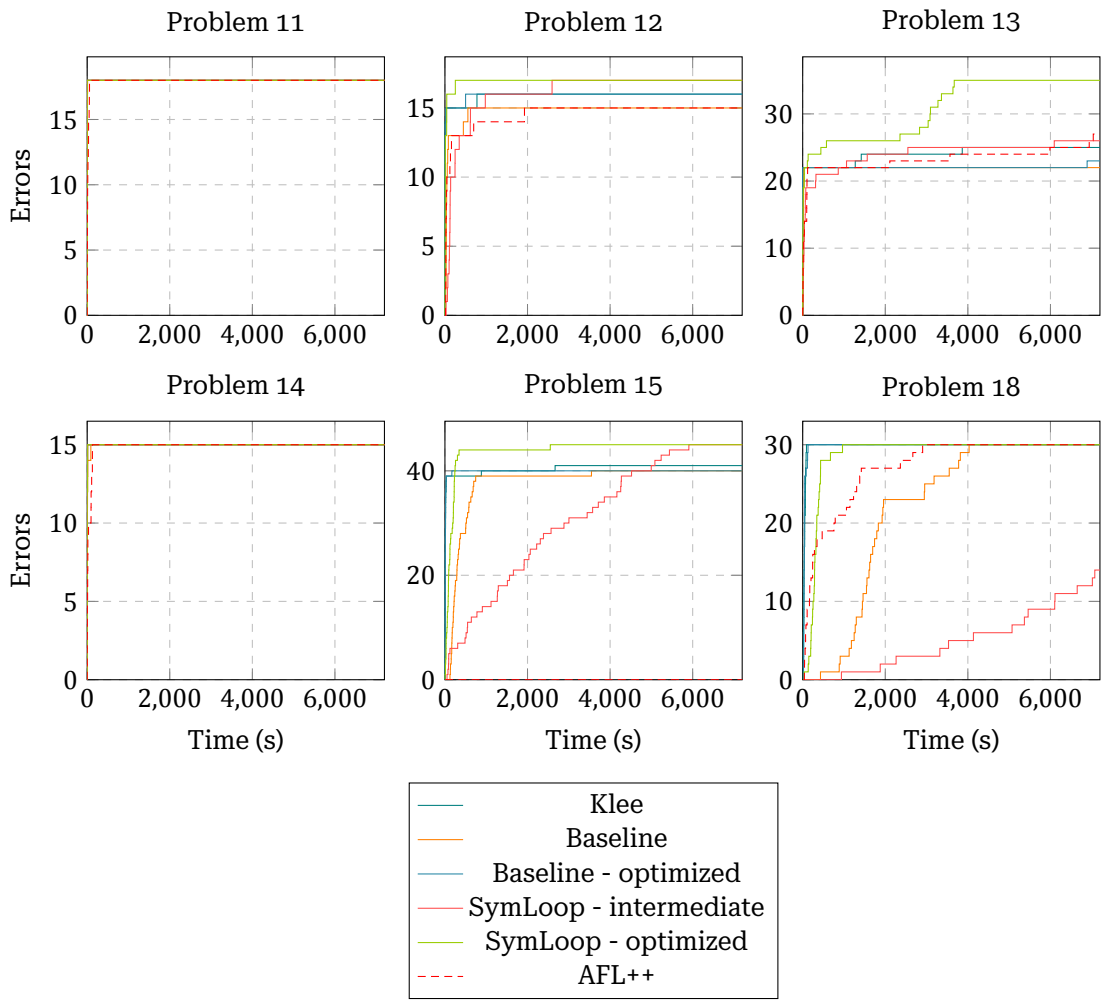


Figure 10.2: Number of errors found over time on the RERS problems. We have ran all methods for 2 hours. The variants of SymLoop all use a detection depth of 10 and a loop unroll amount of 50.

errors SymLoop did not find within the same time frame. On problem 13, error 64 was found by Klee and the baseline, but not by SymLoop . The only other case where this occurred is on problem 15, with error 1.

Although finding more errors can be a goal by itself, this might be achievable with performance optimizations alone. Instead, we view our method as capable of efficiently finding a new class of errors. Our method specifically focuses on reaching new code (and thus errors) that requires many iterations through loops. As shown by the two cases on problem 13 and problem 15, running a traditional symbolic execution method alongside our symbolic execution can be necessary to find more errors in the same time frame.

10.3 Active 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 ‘W1’ for W-method with depth 1, and ‘Symb D10L50‘ respectively. We have run the same experiment using a W-method with a depth of 10 and the Loop-W-method. We let the methods run for 5 days before stopping them manually.

Table 10.2 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

	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 10.2: 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.

to find counterexamples for the hypothesis. As 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.

Overall, we have shown that treating loops in models differently from other transitions in hypothesis models can lead to more complete models of the system.

11 Conclusion

This chapter lists our conclusions and answers the research questions proposed in Section 1.2. We also discuss the overall outcome of the research and note areas for future work.

11.1 Research Questions

In the introduction, we started this thesis with the following research question:

How can you extend symbolic execution to reach states that require repetitive iterations through loops?

During this thesis, we have answered all of our research questions and developed a new symbolic executor with the ability to detect and use loops in the input. By implementing the method and running it on the RERS challenges, we have shown that this method works and is able to quickly reach code that requires many iterations through loops. Below, we repeat our sub-questions and reiterate how our method works.

How can loop structures in symbolic execution be detected?

We have shown different techniques for loop detection in Chapter 5. The stack-based method can always be applied, but in our work we used the retrieval of input as a detector for loop iterations. This method works due to our assumptions on the execution model.

We introduced the notion of loop path extension, which allows us to check whether a certain loop path can be executed again. This check can be performed by checking for satisfiability of the extended loop constraint in conjunction with the current path constraint. Additionally, we proposed a method for detecting self loops with no changes to the internal state.

Is it possible to generalize the assignments in loop structures into constraints for a symbolic executor?

In Chapter 6, we showed generalization for loop structures with simple assignments. Using these generalizations, we were able to create constraints for variables that were reassigned in the loop. Some initial experiments proved that this method was a solution for loop structures with simple assignments. However, this method was not able to deal with arbitrary expressions in the loop. To create constraints for arbitrary loop paths, we introduced the concept of unrolled loop constraints.

How can loop structures be used to reach new branches in symbolic execution?

By creating an iteration constraint that represents taking one iteration in the unrolled loop constraints, symbolic execution can continue after creating the iteration and unrolled loop constraints. Finding inputs for any future branch constraint allows any of the loop iterations to be used. Creating the iteration and loop constraints allows our symbolic executor to satisfy branches with any of the iterations through the loop with just one call to the solver. When a branch is satisfiable, our technique generates an input which contains the exact input that executes the loop the required

number of times to visit the targeted branch. The solver checks many paths through a loop at the same time, where state-of-the-art methods only consider one path per call to the solver.

Does detecting and using loop structures allow symbolic execution to cover more states of a program?

We have verified our symbolic executor on the RERS Problems. Even though these problems are heavily obfuscated, our symbolic executor is able to find errors in these programs that state-of-the-art tools are not able to find. The inputs that triggered these new errors showed long looping patterns, signifying the ability of our symbolic executor to cover more states in programs. We outperform state-of-the-art fuzzing and state-of-the-art symbolic execution.

How can we use symbolic execution for equivalence oracles in active learning?

Current methods for equivalence checking in active learning do not handle loops differently than other transitions. In some systems, these loops can show different behavior after a number of iterations through them. We propose loop equivalence checkers for checking the behavior of the system after many iterations through loops. In black-box scenarios, the Loop-W-method can be applied. For systems where path constraints can be collected, the SymLoop equivalence checker can verify the behavior and guarantee equivalent behavior for some self-loops. Unrolling the path constraint allows detecting changes for distinguishing traces when following the loop once in comparison with following the loop multiple times. For systems where running membership queries is costly, symbolic execution reduces the cost associated with model checking. Additionally, symbolic execution can detect changes in paths through the system, whereas the Loop-W-method can only detect changes through the output. Our complexity analysis and results show that these special methods for loops are significantly more efficient at finding counterexamples for loops than the traditional W-method.

11.2 Future Work

In this thesis, we implemented our method only for a single execution model. To apply our symbolic execution technique to any program, we suggest adding the stack-based detection for loops.

Additionally, we showed generalized expressions for some specific assignment expressions in Section 6.1.3. In our implementation, we did not use these generalizations, as unrolling loops also handles these cases and even handles more. However, the generalizations can be more efficient than loop unrolling, as they result in smaller models for the solver. Additionally, when more iterations than the unroll amount are required, applying these generalizations can help reach those areas of the code faster.

Bibliography

[1] Dana Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and Computation* 75.2 (Nov. 1987), pp. 87–106. ISSN: 08905401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://linkinghub.elsevier.com/retrieve/pii/0890540187900526 (visited on 09/23/2022).

[2] Thanassis Avgerinos et al. “Automatic exploit generation”. In: *Communications of the ACM* 57.2 (2014). Publisher: ACM New York, NY, USA, pp. 74–84.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[4] Cristian Cadar et al. “EXE: Automatically generating inputs of death”. In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008). Publisher: ACM New York, NY, USA, pp. 1–38.

[5] Sofia Cassel et al. “Extending Automata Learning to Extended Finite State Machines”. In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*. Ed. by Amel Bennaceur, Reiner Hähnle, and Karl Meinke. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 149–177. ISBN: 978-3-319-96562-8. DOI: 10.1007/978-3-319-96562-8_6. URL: https://doi.org/10.1007/978-3-319-96562-8_6 (visited on 09/28/2022).

[6] Tom Catshoek. “Exploiting structure in counterexamples to speed up equivalence checking in the minimally adequate teacher framework: Active Learning”. In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3A8e5486e4-da37-43ac-8653-3f1e87a95253 (visited on 09/20/2022).

[7] T.S. Chow. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978). Conference Name: IEEE Transactions on Software Engineering, pp. 178–187. ISSN: 1939-3520. DOI: 10.1109/TSE.1978.231496.

[8] Joeri De Ruiter and Erik Poll. “Protocol state fuzzing of TLS implementations”. In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 193–206.

[9] Andrea Fioraldi et al. “AFL++ combining incremental steps of fuzzing research”. In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 2020, pp. 10–10.

[10] Robert M. Hierons et al. “Using adaptive distinguishing sequences in checking sequence constructions”. In: *Proceedings of the 2008 ACM symposium on Applied computing*. SAC ’08. New York, NY, USA: Association for Computing Machinery, Mar. 16, 2008, pp. 682–687. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1363850. URL: https://doi.org/10.1145/1363686.1363850 (visited on 02/27/2023).

[11] Falk Howar, Bernhard Steffen, and Maik Merten. “Lessons learned in the ZULU challenge”. In: (), p. 18.

[12] Falk Howar et al. “The RERS challenge: towards controllable and scalable benchmark synthesis”. In: *International Journal on Software Tools for Technology Transfer* 23.6 (2021). Publisher: Springer, pp. 917–930.

[13] Stefan Huster et al. “Efficient Testing of Different Loop Paths”. In: *Software Engineering and Formal Methods*. Ed. by Radu Calinescu and Bernhard Rumpe. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 117–131. ISBN: 978-3-319-22969-0. DOI: 10.1007/978-3-319-22969-0_9.

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

[15] Marc Jasper et al. “RERS 2019: Combining Synthesis with Real-World Models”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 101–115. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_7.

[16] Kasper Luckow et al. “JDart: A Dynamic Symbolic Analysis Framework”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 442–459. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_26.

[17] Maik Merten et al. “Next Generation LearnLib”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 220–223. ISBN: 978-3-642-19835-9. DOI: 10.1007/978-3-642-19835-9_18.

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

[19] Jan Obdržálek and Marek Trtík. “Efficient Loop Navigation for Symbolic Execution”. In: *Automated Technology for Verification and Analysis*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 453–462. ISBN: 978-3-642-24372-1. DOI: 10.1007/978-3-642-24372-1_34.

[20] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with ${$SymCC$}$: Don’t interpret, compile!” In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 181–198.

[21] 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. New York, NY, USA: Association for Computing Machinery, Sept. 5, 2005, pp. 62–71. ISBN: 978-1-59593-148-1. DOI: 10.1145/1081180.1081189. URL: https://doi.org/10.1145/1081180.1081189 (visited on 02/24/2023).

[22] Prateek Saxena et al. “Loop-extended symbolic execution on binary programs”. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. 2009, pp. 225–236.

[23] Muzammil Shahbaz and Roland Groz. “Inferring Mealy Machines”. In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 207–222. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_14.

[24] Vaibhav Sharma et al. “Veritesting Challenges in Symbolic Execution of Java”. In: *ACM SIGSOFT Software Engineering Notes* 42.4 (Jan. 11, 2018), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/3149485.3149491. URL: https://doi.org/10.1145/3149485.3149491 (visited on 12/08/2022).

[25] Frits Vaandrager et al. “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*. Springer, 2022, pp. 223–243.

Glossary

conjecture A question in the minimally adequate Teacher framework that a Learner asks the Teacher on whether a given hypothesis matches the regular set the Learner is learning. 11

extended loop constraint Constraint that represents executing the loop one time more than the loop constraint. 20, 22, 24, 25, 29, 45

generalized loop constraint Constraint that represents executing the loop an arbitrary number of times. This loop constraint can only be created for simple loop paths. 22

iteration constraint Constraint that represents assigning the variables to one of the iterations in the extended loop constraint. 26, 27, 29, 31, 38

L* The L* learning algorithm introduced by [1] to learn state machines of regular language 11–13

LM* The LM* learning algorithm introduced by [23] to learn Mealy Machines 13

loop constraint Constraint generated when taking one or multiple iterations through a loop. 20, 22, 24, 29, 45

membership query A query in the minimally adequate Teacher framework that a Learner asks the Teacher on whether a given string *t* is a member of the regular set that the Learner is learning. 11

path constraint In symbolic execution, a path constraint is a conjunction of constraints that represent following the current execution path. The path constraint contains both the assignment to variables as the conditions that the execution path took. 20, 29

self loop constraint Constraint that checks whether a path through a loop is a self loop that keeps all the variable in the same state as before executing this path 20, 36

unrolled loop constraint Constraint that represents executing the loop an arbitrary amount up to a certain limit, by unrolling the assignments in the loop. This type of constraint can be created for any loop path. 24–27, 29, 31, 32, 35, 36, 38

Acronyms

AVD Automated Vulnerability Discovery 7

CFG Control Flow Graph 9, 19, 21

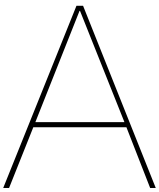
DFA Deterministic finite automaton 10

LTL Linear temporal logic 13

MAT minimally adequate Teacher 11, 34, 45

SMT Satisfiability Modulo Theory 8, 10

SSA Static Single Assignment 8



RERS Results

In the tables below we have included the time it takes to find errors on each RERS problem. Each row in a table represents a different solution. The column `m` indicates the time limit for running that specific solution. The column labeled `#err` contains the number of errors found. Each column after `#err` shows a specific error. The numbers in the table are the time it took to find the specific error. A minus (-) denotes that a solution did not find the error within the specified time.

Program	d	l	m	#err	0	9	13	24	26	39	42	48	52	60	62	74	75	82	91	93	94	95
afl			120	18	2	6	19	53	6	11	33	11	56	1	1	34	5	48	28	2	15	0
baseline	0		10	18	1	3	6	4	3	5	8	2	10	1	3	2	7	3	2	1	5	0
baseline-long	0		120	18	1	3	6	4	3	6	8	2	4	1	3	2	8	3	3	1	5	0
baseline-optimized	0		120	18	0	1	1	1	1	1	1	0	2	0	1	0	1	0	0	0	1	0
klee			120	18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
loop-sym-initial	10	50	10	18	1	4	7	5	4	6	9	2	6	1	3	2	8	3	3	2	5	0
loop-sym-initial	5	10	10	18	1	4	7	5	5	7	9	2	6	1	3	2	9	3	3	2	5	0
loop-sym-intermediate	10	50	120	18	1	3	5	4	4	5	7	1	9	1	3	2	7	2	2	1	4	0
loop-sym-optimized	10	50	10	18	0	1	1	1	1	2	1	0	1	0	1	0	1	1	1	0	1	0
loop-sym-optimized	10	50	120	18	0	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	0

Table A.1: Results from running on Problem 11.

Program	d	l	m	#err	3	6	8	10	25	26	28	37	56	65	68	71	73	74	76	77	79
afl			120	15	22	1926	120	18	134	697	25	21	-	32	-	35	21	47	158	14	19
baseline	0		10	14	63	-	31	18	34	432	28	29	-	20	-	64	27	36	67	25	23
baseline-long	0		120	15	29	556	67	57	41	446	83	46	-	35	-	30	19	34	33	8	28
baseline-optimized	0		120	16	2	27	2	1	1	21	4	2	500	1	-	2	2	1	7	1	1
klee			120	16	2	4	1	1	2	23	2	2	776	1	-	2	2	2	3	1	1
loop-sym-initial	10	50	10	12	181	-	277	170	-	-	196	200	-	114	-	205	188	135	221	74	161
loop-sym-initial	5	10	10	13	119	250	99	112	91	-	127	129	-	72	-	130	123	80	-	65	117
loop-sym-intermediate	10	50	120	17	244	138	108	117	101	614	125	126	618	60	2595	246	346	71	978	28	122
loop-sym-optimized	10	50	10	17	5	6	7	7	16	38	8	9	38	3	156	5	7	3	6	1	4
loop-sym-optimized	10	50	120	17	6	7	8	8	42	44	9	11	44	3	252	6	8	4	6	2	5

Table A.2: Results from running on Problem 12.

Program	d	l	m	#err	1	3	10	11	14	16	23	30	31	34	35	37	45	47	48	49	52	60	61	62	64	65	68	70	71	72	74	75	79	82	83	86	87	90	94	97
afl			120	27	3	43	1	-	-	17	97	10	97	6	9	42	-	5987	119	6	33	-	-	6939	-	7037	104	7	5	98	13	14	-	101	3567	-	103	105	-	2110
baseline	0		10	22	5	5	7	-	-	20	12	17	15	1	3	15	-	-	15	6	24	-	-	-	-	-	13	8	4	22	9	7	-	22	-	-	8	36	-	-
baseline-long	0		120	22	11	6	7	-	-	23	14	19	21	1	3	17	-	-	4	6	27	-	-	-	-	-	20	8	5	25	10	7	-	24	-	-	11	13	-	-
baseline-optimized	0		120	23	2	0	1	-	-	1	2	2	1	0	0	1	-	-	1	0	2	-	-	-	6888	-	1	1	0	1	1	1	-	1	-	-	1	1	-	-
klee			120	25	1	1	1	-	-	1	1	1	2	1	1	1	-	-	2	1	1	3865	-	-	1273	-	1	1	1	5	1	1	-	2	-	1418	1	1	-	-
loop-sym-initial	10	50	10	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-sym-initial	5	10	10	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-sym-intermediate	10	50	120	26	75	10	13	-	1062	18	32	321	61	1	8	40	-	6092	41	12	2548	-	-	-	-	861	42	17	67	65	312	33	-	64	-	1563	22	29	-	-
loop-sym-optimized	10	50	10	26	2	1	4	-	148	9	3	5	4	0	1	3	-	134	10	1	41	-	-	-	-	122	10	2	10	5	5	8	-	5	-	485	2	3	-	-
loop-sym-optimized	10	50	120	35	2	1	3	3095	132	7	41	38	12	0	1	42	3276	576	3	1	9	3670	3038	3361	-	113	9	1	1	12	4	6	3084	7	2356	440	2	3	2830	3639

Table A.3: Results from running on Problem 13.

Program	d	l	m	#err	4	15	18	25	27	28	36	41	59	60	73	76	77	90	97
afl			120	15	95	33	2	126	108	6	12	1	15	10	4	17	2	93	127
baseline	0		10	15	12	6	1	8	7	1	4	0	2	3	2	5	1	6	4
baseline-long	0		120	15	12	6	1	8	7	1	4	0	89	3	2	5	1	6	10
baseline-optimized	0		120	15	2	1	0	1	1	0	1	0	1	1	0	9	0	1	1
klee			120	15	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
loop-sym-initial	10	50	10	15	186	9	1	11	10	2	6	0	167	4	1	7	1	8	179
loop-sym-initial	5	10	10	15	16	8	1	11	9	2	6	0	150	4	3	7	1	8	14
loop-sym-intermediate	10	50	120	15	13	6	1	8	7	1	4	0	2	3	2	5	1	6	11
loop-sym-optimized	10	50	10	15	15	1	0	1	1	0	1	0	1	1	0	13	0	1	12
loop-sym-optimized	10	50	120	15	10	1	0	1	1	0	10	0	1	1	0	12	0	1	1

Table A.4: Results from running on Problem 14.

Program	d	l	m	#err	1	4	7	8	9	12	15	16	17	20	21	24	25	27	28	30	31	33	34	35	38	39	44	46	49	53	54	60	63	64	68	71	72	74	76	78	80
afl			120	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
baseline	0		10	36	-	380	264	189	196	303	426	-	-	101	285	470	252	189	321	-	412	230	491	465	292	-	198	207	103	-	413	203	-	-	-	170	335	364	388	229	271	-
baseline-long	0		120	40	3549	305	684	166	627	255	131	290	-	506	239	583	212	165	577	-	521	203	350	501	247	-	170	178	356	-	330	175	-	744	-	151	122	294	548	202	225	-
baseline-optimized	0		120	40	168	10	11	5	5	16	16	29	-	25	11	6	7	5	8	-	17	6	11	9	8	-	17	5	11	-	10	5	-	9	-	4	12	9	30	6	7	-
klee			120	41	882	5	7	4	9	15	6	29	-	5	7	9	6	6	4	-	33	17	28	8	6	-	6	7	5	-	5	16	-	6	2668	7	6	11	7	22	15	-
loop-sym-initial	10	50	10	4	-	-	-	-	-	307	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	553	-	-	-	-	-
loop-sym-initial	5	10	10	7	-	-	-	466	-	-	-	-	-	-	-	-	-	462	-	-	-	-	-	-	-	-	82	-	-	-	516	-	-	-	337	-	-	-	-	-
loop-sym-intermediate	10	50	120	45	-	2221	1292	487	512	1659	4257	5237	3576	1272	1513	639	1116	91	4996	3861	2570	906	2883	5085	1564	3715	51	548	92	5437	4267	542	3445	1919	5907	4164	1916	3003	2310	73	122
loop-sym-optimized	10	50	10	44	-	94	213	20	21	64	12	89	145	48	56	12	9	20	71	183	4	28	299	83	59	167	22	248	125	205	114	23	-	276	308	190	82	90	97	212	50
loop-sym-optimized	10	50	120	45	-	88	121	28	230	205	206	83	176	256	71	223	49	28	12	199	4	41	115	79	122	187	142	29	137	303	225	29	2551	78	342	19	12	85	91	120	57

Table A.5: Results from running on Problem 15.

Program	d	l	m	#err	0	2	11	12	19	22	23	25	31	35	37	46	50	51	53	56	61	63	66	69	72	76	78	79	85	86	95	96	97	99
afl			120	30	223	2	78	73	91	165	6	60	63	69	77	59	60	73	60	71	8	73	2	60	62	118	82	92	104	178	63	62	77	66
baseline	0		10	30	8	492	1	23	568	19	4	6	19	5	14	566	12	516	1	16	4	24	3	2	21	14	9	23	17	10	22	7	493	439
baseline-long	0		120	30	9	574	515	26	552	22	4	6	21	5	14	16	13	25	1	18	5	28	3	2	23	15	1	26	19	10	24	7	553	8
baseline-optimized	0		120	30	1	76	2	4	77	3	84	1	3	1	2	2	2	4	0	3	1	4	1	1	3	2	2	4	3	2	1	1	77	1
klee			120	30	2	2	2	2	4	3	2	2	2	2	6	3	2	2	2	2	2	3	2	1	9	3	2	4	3	2	5	2	2	2
loop-sym-initial	10	50	10	24	9	-	-	27	-	23	4	6	22	5	15	17	-	25	1	18	5	29	3	2	24	15	1	27	20	11	-	7	-	8
loop-sym-initial	5	10	10	24	9	-	-	27	-	23	4	6	-	5	15	17	14	26	1	18	5	29	3	2	-	16	1	27	20	11	25	7	-	8
loop-sym-intermediate	10	50	120	30	7	533	276	23	535	20	3	5	19	4	13	14	11	21	0	15	4	24	2	2	590	13	1	23	17	9	5	6	533	7
loop-sym-optimized	10	50	10	30	1	87	78	4	83	3	1	1	3	1	2	2	2	3	0	2	1	4	1	1	3	2	0	4	3	2	1	1	83	1
loop-sym-optimized	10	50	120	30	1	71	1	3	71	2	75	1	2	1	2	2	1	3	0	2	1	3	0	0	2	2	1	3	2	1	1	1	71	1

Table A.6: Results from running on Problem 17.

Program	d	l	m	#err	0	1	5	7	12	14	19	23	25	26	29	30	31	34	48	52	53	62	64	65	66	71	72	73	77	79	83	
afl			120	30	108	59	2667	1307	41	1232	172	244	166	2366	472	201	1060	69	45	70	107	758	164	241	44	241	1420	347	285	1145	78	13
baseline	0		10	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	426	-	-	-	-	-	-	-	-	-	-	-	
baseline-long	0		120	30	1695	1556	885	1272	3818	1180	3549	1941	3776	1832	1298	1462	1437	1247	2954	431	1794	1606	1645	1740	2946	908	1447	1545	1125	1924	4030	31
baseline-optimized	0		120	30	42	119	20	30	34	28	27	48	33	45	31	36	35	29	49	39	129	39	40	43	36	21	114	38	26	84	40	
klee			120	30	39	69	18	42	92	31	50	65	27	39	15	20	18	13	79	49	65	29	48	82	66	63	27	31	27	70	51	
loop-sym-initial	10	50	10	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
loop-sym-initial	5	10	10	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
loop-sym-intermediate	10	50	120	14	7016	6111	1878	4135	5071	3532	-	-	-	-	-	-	-	-	-	-	-	-	6648	7074	5460	2262	5370	6103	3322	-	938	
loop-sym-optimized	10	50	10	25	-	324	138	249	293	213	-	-	277	407	260	-	300	47	431	334	394	335	-	376	309	148	303	322	200	425	343	2
loop-sym-optimized	10	50	120	30	375	321	134	241	279	207	198	429	270	411	251	290	286	236	433	330	397	964	342	385	297	148	287	320	194	672	341	2

Table A.7: Results from running on Problem 18.

B

Paper Submitted to ICGI 2023

This appendix includes the paper that was submitted to the 16th International Conference of Grammatical Inference (ICGI).

Detecting Changes in Loop Behavior for Active Learning

Anonymous Author

Editor:

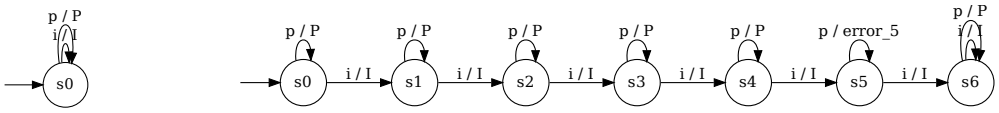
Abstract

In active automaton learning, there is no distinction between transitions that are inside a loop and those that are not. When using automata to model the behavior of software systems, it can be useful to add such a distinction because the system's behavior can change over the number of iterations through these loops. We introduce a loop equivalence checker, which focuses on verifying the behavior of loops in a hypothesis model. It detects changes in behavior after taking many iterations through a loop, requiring very long traces to execute. We propose two methods to test such traces. Our naive approach simply tests each iteration of the loop. Our symbolic approach that uses symbolic execution to check all of these iterations at once. The symbolic approach requires more computation but much fewer queries to the system under test. Additionally, for some loops, the symbolic approach can guarantee executing the loop multiple times has no effect on the behavior. We present results of applying our approaches to problems from the RERS challenge. Compared to traditional equivalence testing methods such as the W-method, our loop checking finds more counterexamples, and thus states, for models with loop dependencies. We also applied our symbolic executor to the RERS challenge, and our method outperforms the state-of-the-art executor Klee by finding more errors.

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

1. Introduction

Active 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). One of the main hurdles for these learning problems is to efficiently find counterexamples to hypothesized models, i.e., to answer 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. After the W-method terminates, it guarantees that if the hypothesis is incorrect, the system under learning has at least w more states than the hypothesis. We propose a new method for answering these queries based on a pattern commonly observed in software systems: loops. Software often contains loops that iterate up to a certain limit. In equivalence checks, such loops are verified in the same manner as any of the other transitions. In software, however, loops tend to behave differently only after a possibly large number of iterations. We investigate the behavior of loops to determine the number of iterations required for a behavioral change. Identifying such a change results in many new states and thus aid the active learning process, see Figure 1.



(*a*) Change not found

(*b*) Change in loop behavior found

Figure 1: 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.

In this paper, we present two methods to test for looping behavior: *Loop-W* a straightforward adaptation of the commonly used W-method (Chow, 1978), and *SymLoop* a method that relies on symbolic execution of an arbitrary number of loop iterations. For the second method, we make use of instrumentation to keep track of the code and paths traversed by a system execution. The first method is fully black box and investigates the hypothesis model to extract looping transitions. We test both methods on problems from the RERS 2020 challenge, which are complex software systems for which the input-output behavior can be represented using (large) Mealy machines. We implement our new methods as equivalence checkers in the LearnLib active learning library (Merten et al., 2011) and demonstrate that our methods improve the performance of active learning on the RERS problems. In addition, by comparing against the state-of-the-art symbolic execution tools Klee (Cadar et al., 2008) and JDart (Luckow et al., 2016), we show that SymLoop's new symbolic execution of loops typically leads to many more code branches being reached when given the same run-time limit. This is a very encouraging result, as both of these state-of-the-art tools use highly optimized interaction with the symbolic solver.

Besides these results, a key contribution of SymLoop is the constraints it uses to efficiently test for loop execution. Our encoding of the looping path constraints makes it possible to test for an arbitrary number of loop iterations using a single call to the symbolic solver. Although this constraint may be large, it is in our experience much more efficient to use a single call than to call the solver again for every additional iteration. As we show below, even on simple code examples, state-of-the-art struggles when they need to solve looping structures.

1.1 Motivation: loops in symbolic execution

There are some limitations in symbolic execution that make the process hard if not infeasible for larger programs. One of the limiting areas is the path explosion problem (Baldoni et al., 2018). This often occurs due to programs making use of looping structures. When running symbolic execution for these structures, each iteration of the loop discovers new paths. With each conditional statement inside the loop, it also creates new decision points.

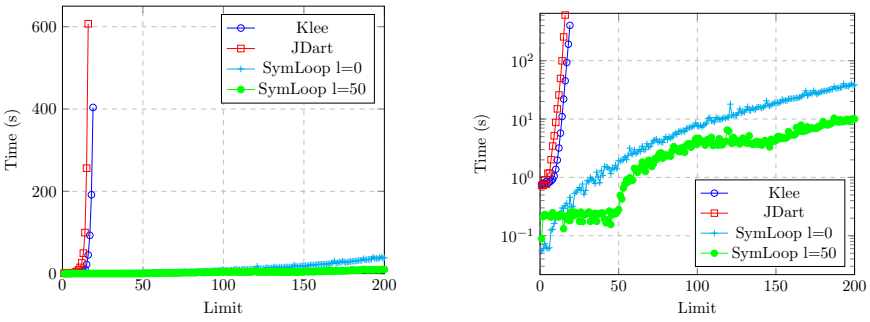
```
int main(int argc, char** args) {
  int limit = atoi(args[1]); // Target to reach
  int i = 0; // Internal state
  int j = 0; // Loop variable
  char symbol; // Character in input
  char* trace = args[2]; // Input: array of symbols
  while ((symbol = trace[j++]) != 0) { // Get next symbol
    if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

Figure 2: C Program used for comparing Klee (JDart uses similar Java code) and SymLoop.

We have tested two different state-of-the-art symbolic execution tools. The C program used for Klee (Cadar et al., 2008) can be found in Figure 2. The Java version for running JDart (Luckow et al., 2016) is semantically equivalent. The program takes an input trace, a series of symbols, and changes the internal state of the program based on this trace. There are two possible input symbols: an `i` or a `p`. The `i` symbol increases the internal variable i by one, whereas the input `p` checks if the i variable is larger than a specified limit. The limit is given as an input argument as well, but is constant during the symbolic run. We use this limit to test the runtime on different input lengths.

The results of Klee and JDart compared to our new method called SymLoop can be found in figure 3. We included two different runs of SymLoop, in the run denoted by $l = 0$, we only skip exploring inputs with self loops and do not unroll the loops. For the run labeled $l = 50$, the detected loops are also unrolled for 50 iterations. Where off-the-shelf symbolic executors like Klee and JDart do not scale, our method is able to reach code that requires many iterations through a loop. When looking at the log scaled plot, we see that the runtime of SymLoop with $l = 0$ grows sub-exponential in terms of the limit, whereas Klee and JDart appear to be exponential for this use case. SymLoop with $l = 50$ is able to find errors more quickly for larger limits. The runtime pattern for $l = 50$ is due to unrolling a loop 50 times being expensive when it is not needed, and cheap when it is.

Many software programs contain loops, and even for a relatively low number of iterations of these loops, symbolic execution takes a considerable amount of time to reach errors. Reaching code and finding bugs that require significantly more iterations of the loop is infeasible. Our main contribution is reaching code that requires more iterations of these loops without inducing a large increase in run time. We expect that adding the SymLoop constructions to state-of-the-art symbolic executors will increase their performance on software containing loops. We leave making such an implementation for future work.



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

Figure 3: Time needed for running symbolic execution on the program from Figure 2 to find an error for a certain limit. For Klee we stopped running past a limit of 19. When JDart crashed past a limit of 16, and did not find the error.

2. Related Work

We consider relevant literature in both active learning and symbolic execution. Symbolic execution is a program analysis technique to test whether certain properties hold for a given program. A program is initially fed with a seeding input. Symbolic execution keeps shadow variables of all the expressions and variables in the program. The inputs to the program are initialized as free variables, to then use a Satisfiability Modulo Theory (SMT) solver to generate new inputs for the program. Each assignment will add a constraint to the path constraint. Assigning to a variable is done through static single assignment, for each new assignment a new shadow variable is introduced. At every branching point along the current execution path, the solver is invoked. Both the path constraint and the (possibly negated) branch constraint are given to the solver. The solver will then yield unsatisfiable if it can not find an input that triggers the opposite path of the branch. If the solver returns satisfiable, a new input can be retrieved from the solver model. When executing this input, it will take the opposite branch compared to the current input. The new inputs are added to a queue and after finishing executing the current input, the next input is executed.

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. Instead of interpreting its LLVM IR, SymCC is implemented as 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. Next section lists 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. Traditionally, symbolic execution creates a path constraint representing a single path that is executed. LESE creates symbolic constraints for variables that represent multiple paths through the loop. In their work, they look for linear relationship between the trip counts. Although the method should work for the example shown in Section 1.1, the use of an input grammar and only considering linear relations are 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. Each loop path or iteration includes reads or writes to different variables. The different iterations are combined to cover different behavior of the loop. If one path only reads from `Var1` and modifies `Var2`, and another path only reads `Var3` and writes to `Var4`, the execution order of these iterations does not influence the final result. 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 section 1.1 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 which require more loop iterations.

Angluin introduced the **minimally adequate teacher (MAT) framework** for learning a deterministic finite automaton (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 learn 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.

TTT was introduced by Isberner et al. to reduce redundancy in the observation table of L^* by keeping track of a discrimination tree and a discriminator trie 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. 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 focussing 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 method such as TTT by needing a comparable number of membership queries. Their method outperforms state-of-the-art algorithms by requiring fewer number of symbols for learning.

Equivalence Methods 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 (A) of each state, all possible sequences of length w over the input alphabet, where one of them is denoted by W and all distinguishing traces of the model, where one is denoted D. These three parts are concatenated, and each sequence $A \cdot W \cdot D$ is checked for equivalence with the model. Checking a single trace is done by resetting the system under learn 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.

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 learn is in the expected state. Additionally, there exists models for which a preset distinguishing set cannot be found.

3. Loop Equivalance Checking

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*. In theory, we only require a single symbolic test, which guarantees the system behaves the same no matter how many iterations the loop is executed. In practice, we split large conjunctions into separate calls to an SMT solver.

3.1 The Loop-W-method

The W-method is not able to efficiently find counterexamples for systems that require multiple iterations through a loop before showing different behavior. To illustrate, let d be the number of inputs that are in the loop, and let n be the number of iterations through the loop before it shows different behavior for one of the distinguishing traces. Let I denote the number of symbols in the input alphabet and s the number of states in the hypothesis. Note that s also denotes the number of access sequences and the number of distinguishing traces. The W-method requires a depth of at least $d \times n$. To find a counterexample for such a loop then requires $s \times I^{d \times n \times s}$ membership queries, with the average at half that amount.

To reduce the amount of queries necessary for equivalence checking for loops, we introduce the Loop-W-method. This method requires an additional parameter l. The parameter l denotes up to which depth, the number of iterations, a looping is checked. For each loop in the hypothesis model and for each distinguishing sequence, the access sequence: A, the loop sequence: L, and the distinguishing sequence D are concatenated. This forms the sequence $A \cdot L \cdot D$. This sequence needs to be formed for all the iterations through the loop up to depth l: $A \cdot L^i \cdot W \cdot D \ \forall \ i \in [1..l]$. If checking any of these sequences results in an output that does not match the hypothesis, it is returned as a counterexample.

3.2 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.

3.2.1 Execution Model

For the symbolic equivalence checker, we assume the execution model of the programs under learn. 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.

3.2.2 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 must be 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.

To detect self loops using symbolic execution, the path constraint and a special self loop constraint can be given as a formula to the SMT solver. The self loop constraint checks whether there are no changes to the internal state. The self loop constraint can be formed as follows. Let $a, b \cdots z$ denote all variables in the internal state. The symbolic shadow variables representing their respective values before executing the loop are $\underline{a}, \underline{b}, \cdots \underline{z}$. With $\overline{a}, \overline{b}, \cdots \overline{z}$ being their symbolic values after the loop (note that for any variable that is not assigned, these are already equivalent). The self loop constraint is defined as $\underline{a} = \overline{a} \wedge \underline{b} = \overline{b} \wedge \cdots \wedge \underline{z} = \overline{z}$. A self loop exists if the path constraint, including the loop constraint, in conjunction with the self loop constraint is satisfiable.

3.2.3 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. With a through z denoting the internal variables, the loop constraint contains expressions over these variables and the input. 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, create an extended loop constraint. The extended loop constraint is created from a copy of the loop constraint. Due to the use of static single assignment, every variable that is reassigned in the loop path needs to be updated to reflect its current value. For example: the loop constraint $i_1 > 1 \wedge k_1 = k_0 + 1$ represents a path with a branch where the input variable i must be greater than 1 and the variable k is increased by one. The extended loop constraint then becomes $i_2 > 1 \wedge k_2 = k_1 + 1$. For the general construction of the extended loop constraint, we introduce some additional notation. Let P denote the loop constraint to be extended. Let $[a/b]X$ denote substituting every a with b in X. And let $\hat{a}, \hat{b} \cdots \hat{z}$ denote the number of reassignments in the loop path to variables $a, b \cdots z$ respectively. The loop constraint can then be updated using $[a_i/a_{i+\hat{a}}, b_j/b_{j+\hat{b}}, \cdots, z_k/z_{k+\hat{z}}]P$ to form the extended loop constraint.

If the current path constraint, including the loop constraint, in conjunction with 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 an arbitrary amount, up to a specified limit l, repeat the substitution process l times and check for satisfiability. This repeated substitution generates the unrolled loop constraint. The unrolled loop constraint is shown in equation 1.

$$\begin{gathered}
[a_i/a_{i+1*\hat{a}}, b_j/b_{j+1*\hat{b}}, \cdots, z_k/z_{k+1*\hat{z}}]P \wedge \\
[a_i/a_{i+2*\hat{a}}, b_j/b_{j+2*\hat{b}}, \cdots, z_k/z_{k+2*\hat{z}}]P \wedge \\
\vdots \\
[a_i/a_{i+l*\hat{a}}, b_j/b_{j+l*\hat{b}}, \cdots, z_k/z_{k+l*\hat{z}}]P
\end{gathered} \tag{1}$$

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 for l leads to generating inputs that trigger different branches. To solve having to decide on the exact number of iterations through the loop, 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. Equation 2 defines this relation.

$$\begin{gathered}
(j = 1 \wedge a_{k+0*\hat{a}} = a_f \wedge b_{k+0*\hat{b}} = b_f \wedge \cdots \wedge z_{l+0*\hat{z}} = z_f) \vee \\
(j = 2 \wedge a_{k+1*\hat{a}} = a_f \wedge b_{k+1*\hat{b}} = b_f \wedge \cdots \wedge z_{l+1*\hat{z}} = z_f) \vee \\
\vdots \\
(j = l \wedge a_{k+(l-1)*\hat{a}} = a_f \wedge b_{k+(l-1)*\hat{b}} = b_f \wedge \cdots \wedge z_{l+(l-1)*\hat{z}} = z_f) \vee
\end{gathered} \tag{2}$$

Variables a_f through z_f represent the symbolic values of variables a through z after executing this loop up to l times. The iteration constraint also includes the variable j to allow easy access to the number of iterations that is required to visit a specific branch after getting a satisfiable model from the solver.

3.2.4 Using SymLoop as equivalence test

Checking for self loops is done by generating the access trace to one of the states in the loop and the subsequent symbols to execute this loop. The access sequence and loop symbols are then run by the symbolic executor. The symbolic executor collects the path constraint from the looping part. This loop path can then be checked for a self loop. If the loop path is a self loop, the model is correct for this loop and no counterexample can be found.

When the loop path is not a self loop, the unrolled loop and iteration constraints are created according to section 3.2.3 to depth l. These are added to the current path constraint. Afterward, 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 in to 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 are indistinguishable up to depth l. For each tested loop, this check calls the symbolic executor once for each distinguishing trace.

4. 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, the LTL problems of 2020 are too simple and can be learned with 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 verify our methodologies for symbolic execution and active learning, we set up three different experiment. In the experiments for symbolic execution, we only focus on getting more coverage, whereas in the active learning experiment, learning models is the goal.

1. Difference in 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.

4.1 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 variables. Z3 (de Moura and Bjørner, 2008) is used to answer any SMT queries. SymLoop detects loops by creating the extended loop constraints, and checks if it is self looping. Traces with self loops are skipped. If the loop is not a self loop, it creates the extended loop constraint and iteration constraint and adds these to the current path constraint. The symbolic execution then continues as normal. Continuing allows future solver queries to use one of the loop iterations. To speed up the symbolic execution, the executor evaluates expressions without symbolic values to constants. 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.

4.2 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 sequence 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 3.2.4. 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.

5. Results

5.1 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 unique errors found. For problem 12, 13, and 15, there are differences. For these problems, we plotted the number of errors found over time in figure 5.1. All methods quickly find the

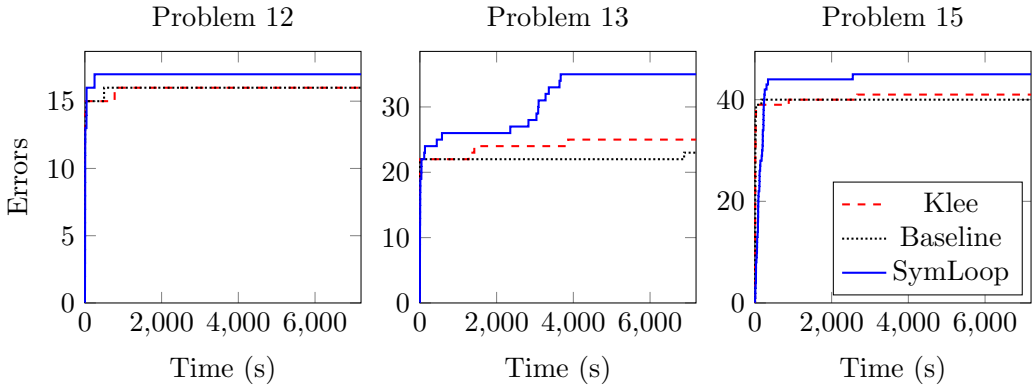


Figure 4: Number of errors found over time on the RERS problems. We have run all methods for 2 hours. SymLoop was run with a detection depth of 10 and a loop unroll amount of 50.

	P11	P12	P13	P14	P15	P17	P18
Baseline	18	16	23	15	40	30	30
Klee	18	16	25	15	41	30	30
SymLoop	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.

bulk of the errors, however SymLoop is able to find more errors. On problem 12, the 17th error is found within 5 minutes, where Klee and the baseline do not find this error within the 2-hour time limit. Although SymLoop finds more errors, on both problem 13 and 15, the baseline and Klee are able to find an error that SymLoop did not find.

5.2 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 the both SymLoop and Loop-W were able to find counterexamples for the 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

	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.

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.

6. Conclusions and future work

We have developed an extension for symbolic execution to cover paths that require many iterations through a loop. Our results for running SymLoop on the RERS challenges of 2020 show that our method is able to find more errors than the state-of-the-art symbolic executor Klee.

Current methods for equivalence checking do not treat loops differently than other transitions. In some systems, these loops can show different behavior. We propose the loop equivalence checker for checking the behavior of the system after many iterations through loops. In black box scenarios, the Loop-W-method can be applied. For systems where path constraints can be collected, the SymLoop equivalence checker can verify the behavior and guarantee equivalent behavior for self-loops. For systems where running membership queries are costly, symbolic execution reduces the cost associated with model checking. Additionally, symbolic execution can detect changes in paths through the system, whereas the Loop-W-method can only detect changes through the output. Our results show that our methods are able to uncover significantly more states when comparing it to the traditional W-method for learning models of the RERS problems.

For symbolic execution for loops, we assumed properties about the execution model. SymLoop was built as a research prototype with the goal of testing our methodology. Future research could generalize the execution model to detect, unroll and create constraints for loops in any software program.

The current learner methods do not handle counterexamples from loops differently from other counterexamples. If a loop must be unrolled to show different behavior after a number of iterations, most of the behavior can be transferred from the existing loop in the model. A learner that uses this knowledge can create a new hypothesis while needing less membership queries for the system under learning.

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.

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.