Bachelor thesis
Computer Science

Radboud University

Checking Model Learning Hypotheses with Symbolic Execution

Author:
Jeremy Guijt
s4063597

First assessor:
prof. F.W. Vaandrager
f.vaandrager@cs.ru.nl

Second assessor:
J. Moerman, MSc
moerman@science.ru.nl

August 22, 2018

Abstract

Model learning can in many ways be helpful to current software development practices, but needs improved efficiency to be used on large systems. Checking conformance of the hypothesized model with the System Under Learning, is still a hard problem. A new method for conformance checking based on symbolic execution is proposed. This method is compared to conformance checking using adaptive distinguishing sequences and mutation-based fuzzing. Mutation-based fuzzing is clearly more efficient, however some possible improvements upon the method are proposed.

Contents

Chapter 1

Introduction

Nowadays, software systems are becoming increasingly complex [23]. This complexity comes at the cost that software developers often cannot comprehend the complete system anymore. Even smaller programs often have too many corner cases or assumptions to be completely comprehended. Model learning can offer a solution. By creating a model of a program, software developers can easily check if the program conforms to its specification [14], analyse its behaviour or find vulnerabilities [11]. In addition, model learning can aid in the automated testing of software [27].

In practice however, it is not feasible to learn models of large systems. The modeling techniques are not efficient enough to quickly model large systems and existing tools are not yet easily applicable [20].

A lot of research attention has been directed towards improving the feasibility of model learning, but determining whether a learned model resembles the subject program (closely enough) or should be refined is still a hard problem. This problem, which is called an Equivalence Query in the Minimally Adequate Teacher framework proposed by Angluin [3], has traditionally been solved by methods with a high running-time complexity and which are only guaranteed to be correct if the model of the subject program is under a given size limit [15]. Recently however, succesfull white box strategies for equivalence checking have emerged [34]. These strategies do not only execute the program to find out its behaviour, but also examine the machine code and/or source code of a program, for example through coverage analysis.

Checking if the learned model conforms to the subject program can be done by repeatedly executing a program with varying input sequences and verifying if the observed responses match the predicted ones. For this process to be as efficient as possible there should be as few as possible test cases

on basis of which can be confirmed (or disproved) that the learned model is correct.

Symbolic execution is a technique where a program is executed with symbolic values and where all possible ways a program can execute are considered by building an execution tree and constraining the symbolic values in each branch. Symbolic execution is a contender for efficient test case generation, since white box methods have an information advantage over black box methods. Additionally, a lot of research has been done on symbolic execution, with a lot of mature tools as a consequence. By concretizing the symbolic values of symbolic execution, test cases for the Equivalence Query can be created in an efficient manner.

This thesis focuses on symbolic execution as a new method for test case generation for equivalence checking. It is compared with adaptive distinguishing sequences [24], a state-of-the-art black box method for equivalence checking that uses a decision tree to distinguish states, and fuzzing [11], which is a bug finding technique that can generate test cases for equivalence checking. Fuzzing can generate test cases through mutating input sequences in a smart way and it is a new and promising white box equivalence checking method.

Chapter 2

Preliminaries

2.1 Model learning

Model learning is the distilling of properties or behaviours out of a subject matter in order to create a model that describes the subject. When applied to software, this can be useful for example to check if an implementations adheres to its specification [14], to compare the behaviour of various pieces of software [31], or to find bugs and vulnerabilities in a program [11].

The subject that learning is applied to is called the System Under Learning (SUL). In other literature the term System Under Test (SUT) is also used.

Many systems (programs, protocols, etc.) can be modeled by finite state machines (FSM). Systems where the output solely relies on the state the system is in and the given input, and which have a finite number of states, can be modeled by Mealy machines and are called reactive systems. A Mealy machine $\mathcal{M}$ is defined as a tuple $(I, O, Q, q_0, \delta, \lambda)$ where I is the (finite) input alphabet, O is the (finite) output alphabet, Q is the set of states, q_0 is the starting state, $\delta : Q \times I \to Q$ is the transition function and $\lambda : Q \times I \to O$ is the output function [26]. Informally, a Mealy machine is a deterministic finite automaton that consumes an input symbol and produces an output symbol on each transition. Figure 2.1 is an example of a Mealy machine.

In this thesis model learning will be applied to reactive systems [18] that can be modeled by Mealy machines. Unlike transformational systems, which are systems that first receive all their input, then go on to process it and finally produce their output, reactive systems are interactive. Reactive systems respond to their inputs and can also be continuous or parallel. Reactive systems are harder to model, as their behaviour cannot be fully described by a mathematical function or relation [33], making it an interesting class

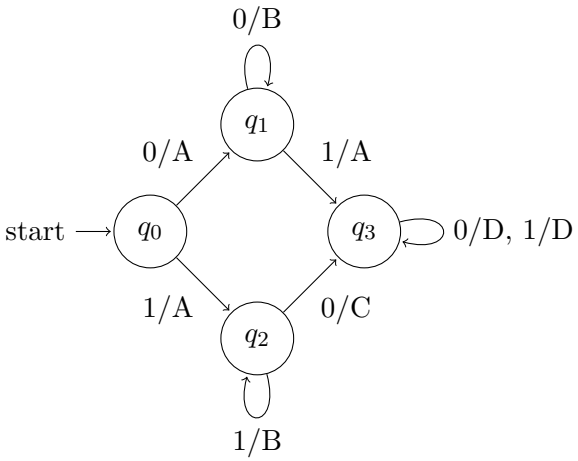


Figure 2.1: A Mealy machine with $I = \{0, 1\}$ and $O = \{A, B, C, D\}$.

of programs for model learning research.

2.1.1 The MAT framework

In this thesis, model learning is done in the minimally adequate teacher (MAT) framework [3].

The MAT framework consists of two components. First, there is an entity called the Learner, which tries to construct a model of the SUL. Second, there exists a Teacher, which has inherent knowledge of the SUL. The Learner doesn't interact with the SUL directly, but gains knowledge on the SUL only through the Teacher. The Learner can do this by asking two types of questions: Membership Queries and Equivalence Queries.

With a Membership Query the Learner asks whether a certain property or behaviour is part of the SUL. It does this by asking if an input sequence is accepted by the machine (for learning deterministic finite automata) or asking what output corresponds with a given input sequence (for learning Mealy machines). Using these Membership Queries the Learner can build a hypothesis of the model.

In the second type of query, an Equivalence Query, the Learner can share its hypothesis with the Teacher and ask for a counterexample. This is called conformance testing. When a counterexample is provided, the Learner starts to refine its hypothesis to incorporate this newfound behaviour. When no counterexample is provided, the Learner considers its hypothesis to be cor-

rect. By iterating this procedure until no counterexample is provided the Learner can create a complete model of the SUL.

2.1.2 Conformance testing

The MAT framework supposes that the Teacher has a sound and complete model of the SUL. In reality however, the Teacher does not have complete information on the SUL, and may not always be able to answer Equivalence Queries correctly. Even in cases where a SUL is implemented based on a specification in the form of a Mealy machine, there is not necessarily a correct model of the SUL, because the implementation may differ from the specification.

To solve this problem, algorithms have been developed which create a test suite based on the hypothesized model of the SUL. The test suite can then be run on the SUL, provided that it is resettable, to check if the model predicts the outcome of all tests correctly. Some algorithms, like the W-method [9] for example, provide a complete test set under the assumption that the minimal number of states in the model is lower than a given bound. I.e., if the model conforms to the SUL for all tests in the suite, either the model is correct, or more states are needed than the given bound. In a black box setting, there is no standardized or correct way of establishing this bound.

The number of tests required for a complete test suite scales rapidly with a growing number of states and a growing input alphabet. The worst case complexity of the W-method and similar methods is $\mathcal{O}(k^{m-n+1}n^3)$, where k is the number of input symbols, m is the number of states in the model underlying the SUL and n is the number of states in the hypothesized model [12].

Another conformance checking method, the adaptive distinguishing sequences (ADS) method [24], is derived from the W-method and relies on finding adaptive distinguishing sequences to generate its test cases. An adaptive distinguishing sequence is a decision tree that is used to distinguish states of a Mealy Machine. In this decision tree, each state can be reached through one sequence of input and output symbols that uniquely identifies that state. It generates smaller test suites than the W-method, but is not always applicable. However, Moerman [28] provides a simple adaptation, the hybrid-ADS method. This method combines ADS with the HSI-method [30]. This combined method can always be applied.

For many realistic use cases however, developing a test suite for check-

ing complete conformance is unfeasible [20]. Possible solutions are using a mapper as an intermediary between the Learner and Teacher [2] or checking for approximate conformance. Using a mapper can greatly diminish the size of the input alphabet by taking a group of inputs for which the behaviour in the SUL is equal and presenting them to the Teacher as just one symbol. For example, suppose a SUL takes an 8-bit int as input, with the behaviour being equal for all inputs less than 10 and the behaviour being equal for all inputs equal to or greater than 10. Naively, there are 256 input symbols that should be evaluated. Using a mapper, a model can be learned using 2 input symbols.

White box conformance testing methods

Methods like the ADS-method or W-method do not use any information about the SUL to work, but base their test suite solely on the provided hypothesis. Such methods, that do not require more information about the SUL than the responses to queries, are called black box conformance testing methods. Contrary to these methods, white box conformance testing methods utilize more information about the SUL to test the provided hypothesis. White box methods can also rely on source code, machine code, control flow graphs, etc., to test the hypothesis.

An example of such a method is mutation-based fuzzing. This technique starts with some inputs (typically the input alphabet in a model learning context) and can mutate this input with a number of rules, such as concatenating two inputs. In the *American Fuzzy Lop*, a fuzzing tool [36], coverage analysis of the binary on which fuzzing is applied can distinguish which inputs cause execution to go in less traversed parts of the binary; these inputs are therefore more interesting. These inputs are saved to a queue and fuzzed again to create more and more "interesting" inputs. In this way, efficient test suites can be generated [22, 34].

Model learning without a mapper is somewhat in between black box and white box model learning. It does not need any information about the internals of the SUL to function, but it may suppose some knowledge of the SUL on beforehand to create the input classes. However, tools that automatically configure the mapper are being researched [1]. Automatic configuration of the mapper would place the technique in the black box category.

2.2 Symbolic execution

Symbolic execution is a technique used for analysing source code and software binaries. The technique is sometimes named differently and is synonymous to terms such as "Automated Binary Analysis". The technique is extensively used in security-oriented research to quickly find bugs or vulnerabilities in examined source code or binaries [32]. Symbolic execution is also used in software development, for example to test if division by zero or a `null`-pointer dereference can occur in the tested software [35].

Symbolic execution on a binary inspects the machine code of a program. In symbolic execution, paths that the execution of a program takes are traced and the corresponding conditions on which the paths are taken are saved. Examples of the uses of symbolic execution are using the paths to create a control flow graph in order to find unwanted behavior, and finding code that is unreachable, through checking if there are actual solutions to the constraints on a path.

For using symbolic execution in this research the `angr` framework was chosen. This framework was chosen because it supports loading binaries (in contrast to tools that require access to source code), it is open source and it has a free license.

2.2.1 How symbolic execution works

Symbolic execution starts with an execution path with only unconstrained symbolic values. The symbolic execution engine will start at the program entry point to step through the program, evaluating all instructions and building the program state on the go.

When an if-statement or loop is encountered, the execution path will branch. In one execution path the condition of the if-statement or loop is true, so that path is constrained with the condition evaluating to be true. Likewise, the other path is constrained with the condition evaluating to be false. Both branches will be stepped through by the symbolic execution engine. The engine can use a selection strategy to try and evaluate more interesting or promising paths over less interesting or promising paths, or it can step through all paths equally.

2.2.2 An in-depth example

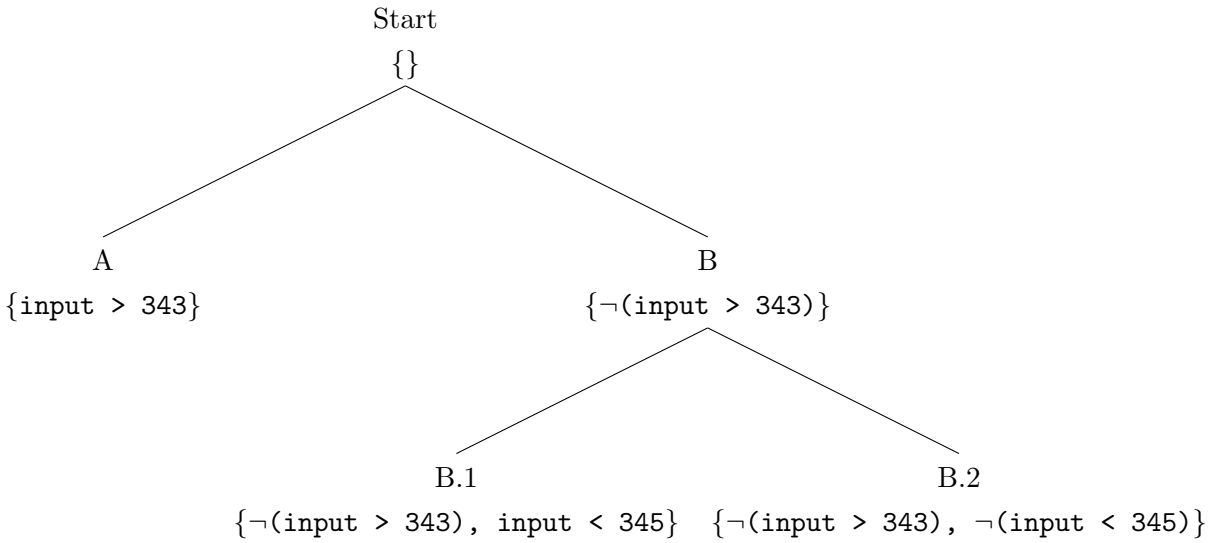
Consider the following code:

```c
#include <stdio.h>
int main()
{
  // Start
  int input;
  scanf("%d", &input);

  if (input > 343) {
    // Path A
    printf("Access denied");
  } else {
    // Path B

    if (input < 345) {
      // Path B.1
      printf("Access granted");
    } else {
      // Path B.2
      printf("Access denied");
    }
  }
}
```

When compiled, this binary will have some equality check of the input variable and the value 343. After the check, a conditional jump will occur. Here the execution path branches. All constraints on the path are copied to both path A and B, while the constraint `input > 343` is added to path A and the constraint `¬(input > 343)` is added to path B. In path B another jump occurs, causing it to split into path B.1 and B.2. Again, constraints are added. For simplification, no paths are taken into account that could stem from the `scanf` or `printf` functions apart from the paths under "normal" execution. See the execution path tree for an overview.



Note that prior knowledge of the source code of the program is unnecessary for using symbolic execution.

2.2.3 angr's inner workings

In its documentation [13], `angr` is described as "a multi-architecture binary analysis toolkit, with the capability to perform dynamic symbolic execution and various static analyses on binaries." Under the hood, the framework consists of multiple modules, most notably: `angr` (the main module), `CLE`, `Claripy` and `PyVEX`.

Analysis of a binary starts with loading the binary, which happens through the `CLE` module. Since a binary uses addresses to enter or return from functions, it is necessary to map the binary on a virtual address space. Because `angr` can load binaries from multiple architectures, an abstraction from the architecture is needed to perform analyses. For this, `angr` uses the VEX intermediate representation that was created for Valgrind, a framework for building dynamic analysis tools [29]. VEX abstracts the differences in architecture, like register names, and provides an easy way for the rest of the framework to interact with the binary. The `PyVEX` module exposes the intermediate representation to the other modules.

`angr` emulates execution of a binary by providing a blank program state and stepping through the intermediate representation, all the while updating the program state. In contrast to normal execution, where each step results in precisely one successor state, a step in symbolic execution can produce multiple successor states. A simulation manager keeps track of all successor states and groups active, deadended en errored states. During symbolic execution, active states are stepped through until they either finish execution or raise an error.

At the start of the emulation, the program state will contain uninitialized memory, registers and so forth. However, during emulation concrete and symbolic values will be added to the program state, for example by loading a value that's compiled in the binary into a register (a concrete value) or by reading the contents of a file (an unconstrained, symbolic value). All these values are implemented as Abstract Syntax Trees (AST) by `angr`'s solver engine `Claripy`. Operations on the values, like incrementing a symbolic value, are saved in the AST. Finally, constraints on the program state that are found during execution, for example at an if-statement, are saved in the solver engine.

The solver engine uses Z3 [10], an SMT solver developed by Microsoft, in its backend. The solver engine can determine if the constraints of a state are satisfiable (i.e., determine if the state is reachable), it can check if a given concrete value is a solution of a symbolic value (taking into account all constraints on the state), and it can produce concrete solutions for symbolic values.

2.2.4 Challenges of symbolic execution

Symbolic execution can work wonders in automated bug searching and program verification [8, 4], but it still needs to overcome some challenging problems. The two main problems of symbolic execution are described below.

Path Explosion

Symbolic execution works by collecting all constraints on the program state on a given execution path. Programs often use loops to iterate a part of the code until a certain condition is valid. Since there is a check on the condition in each iteration of the loop, there will be a lot of branches when the program is symbolically executed. Since loops can iterate indefinitely, the number of branches is infinite. Because of this it can be unfeasible to analyze a program with symbolic execution when it uses loops, because of the large number of branches.

Attempts at reducing path explosion have been made, for example with state merging, where paths can be merged if the program state and next instruction are the same. However, loops are often used to change the state of the program on each iteration. Recently progression has been made in analyzing programs suffering from path explosion by Avgerinos et al. [4]. In the paper dynamic analysis (i.e. symbolic execution) and static analysis are interchanged. The static analysis (termed static symbolic execution in the paper of Avgerinos et al.) creates a logical formula of parts of the programs code. By employing static analysis on loops the path explosion problem of symbolic execution can be alleviated. For example, the node coverage, which measures the lines of code covered by symbolic execution, improved 11.7 percentage point when using this new technique compared to "classic" symbolic execution with the coreutils library of Debian as subject programs.

Further research is needed to make symbolic execution more feasible for all programs.

SMT solving

Each execution path has its state modeled by constraints. Satisfiability and example outputs are typically produced by SMT solvers. SMT is a decision problem for the satisfiability of an expression [6]. This expression is a logic formula that can have all kinds of terms, which are formed through so-called theories. Examples of theories are linear arithmetic and array theory.

SMT-solving is an undecidable problem if the expression contains non-linear functions (e.g. exponential functions). Nevertheless a lot of effort has been put into creating efficient and powerful SMT-solvers. Nowadays there are some SMT-solvers that can solve many, but not all, expressions quickly. Further improvements to these SMT-solvers would make symbolic execution more feasible as an analysis tool for large programs.

2.3 RERS challenge

The Rigorous Examination of Reactive Systems (RERS) [16] challenge is a challenge to solve reachability questions and LTL formulas on benchmarks created for the challenge. The aim of the challenge is to compare various methods to answer the questions. The benchmarks all have similar behaviour: the program waits for an input number, does some computations and it produces an output number. The program does this until the program ends or an error occurs.

The RERS benchmarks are the main benchmarks used in this thesis. Because of the way all benchmarks work, their behaviour can be modeled by Mealy Machines quite well. In this thesis both the reachability problems and LTL-problems from RERS 2016 are used, counting 18 benchmarks in total.

Chapter 3

Research

3.1 Theoretical improvement of symbolic execution on equivalence checking

In this section an explanation is given of why using test cases generated by symbolic execution could be a very efficient conformance testing method. The described improvement is theoretical and is an overestimate of the actual improvement. Nevertheless, the described improvement may still be close to the actual improvement when symbolic execution is applied to programs found in practice.

Model learning algorithms ensure that all input symbols are tested for each state of a hypothesis. This means that all input symbols that can "trigger" a given edge in the model underlying the SUL are discovered during the learning phase, if the hypothesis has found all nodes of the model underlying the SUL. Therefore, when doing equivalence checking it is sufficient to have only one test case that triggers this edge. Even if there are more test cases that are equal except for the last input symbol and that can trigger the same edge in the hypothesis, it is unnecessary to test them during equivalence checking. This is because the model learning algorithm will have made sure that the behaviour is already supported in the hypothesis, so using more of these test cases can not help producing a counterexample.

This shows that a complete test suite can be created with as many test cases as there are edges in the model underlying the SUL.

In many programs, there are a lot of inputs that cause the program to behave the same way. Black box equivalence checking techniques will not pick up on this, because they don't have information about which inputs cause identical behaviour until after they have tried all inputs. Since white

box techniques can extract this information from source code or machine code, white box techniques can pick up on this. Because of this white box techniques can use fewer test cases to fully test the equivalence of the hypothesis and the SUL.

Now we will go on to show under which assumptions symbolic execution can create a complete test suite for a SUL.

For a moment, let us assume that symbolic execution can produce all possible execution paths of a program and that the SMT solver can either solve a given set of constraints or show that they are not satisfiable (SMT solvers are NP-complete, but we'll forego this for the sake of the argument). Additionally, symbolic execution should identify and merge states it has earlier encountered by checking if all constraints and the instruction pointer are equal. This will prevent duplicate symbolic execution of the code and duplicate execution paths.

In efficiently programmed programs, various inputs that produce identical output when the program is in the same state will be in the same execution path. This means that when the input symbols and output symbols belonging to an execution path are concretised, there is no other execution path that can have the same input and output symbols when concretised. Now each execution path corresponds with precisely one path in the model underlying the SUL, because all execution paths show unique behaviour. Furthermore, because the execution paths contain all possible behaviours of the program, the corresponding paths in the model underlying the SUL will also exhibit all possible behaviours of the program.

If we use the the concretised inputs for all execution paths, we can generate test cases that elicit all possible behaviours of the program. This means that every edge in the model underlying the SUL is covered by the test suite. This in turn means that we can check if all edges of the model underlying the SUL are present in the hypothesis using this test suite.

This shows that in the situation that the assumptions made earlier are true, symbolic execution could drastically reduce the number of queries needed for Equivalence Checking of a model (namely, to the number of edges of the model underlying the SUL). In reality however, symbolic execution will not always be able to deduce all execution paths and SMT solvers can't solve all constraints. In addition, there may be various execution paths that that, in the same state, map different input symbols to the same output symbol (caused by an ineficciently programmed SUL). However, even with

these imperfections, there may be a worthwhile reduction in the number of test cases for equivalence checking when using symbolic execution, especially in programs with large numbers of input symbols and few execution paths.

3.2 Experimental setup

The aim of the research is to evaluate if using symbolic execution for answering the equivalence queries during model learning is beneficial compared to other strategies for answering equivalence queries. As such, the experiments were carried out in three groups. The first group used symbolic execution to create test cases for the Equivalence Oracle (the part of the learning setup that answers Equivalence Queries). The second group used fuzzing to create test cases for the Equivalence Oracle. The third group used adaptive distinguishing sequences for test case generation to act as a baseline for the other two groups.

3.2.1 RERS benchmarks

The benchmarks from the RERS challenge of 2016 were used. There are 18 benchmarks in total; 9 benchmarks are LTL problems and 9 benchmarks are reachability problems. The parallel problems of the challenge were not considered. In the challenge, the objective of the LTL benchmarks is to test validity of certain LTL-expressions, while for the reachability benchmarks the objective is to check which errors in the source code are reachable. However, in this paper the objective for both types of benchmark is to learn a Mealy machine of the benchmarks behaviour.

The LTL and reachability benchmarks are categorized by difficulty of the problem and program size. Each of the categories “plain, simple”, “arithmetic, medium” and “data structure, hard” has a small, medium and large benchmark. Four benchmarks have an input alphabet ranging from 1 to 5, six benchmarks have an input alphabet from 1 to 10 and the remaining eight benchmarks have an input alphabet ranging from 1 to 20.

All program behaviour is as follows: The program loops, waiting for input and producing an output number for each input number. On each iteration of the loop the program first reads an int as input, it changes the state of the program and an output symbol is printed. If the entered number is not in the input alphabet, the program stops execution. The state is kept through a number of variables. The state is changed upon input by conditionally assigning other values to the state variables and printing an output symbol. If the state isn’t changed, the program prints a message

saying the input was invalid and continues the loop.

3.2.2 Test case generation with symbolic execution

To symbolically execute each of the RERS benchmarks, `angr` was used. Symbolic execution began at the main entrypoint of the program and used a breadth-first strategy to step through all execution paths. Execution paths were collected in "live" and "deadended" stashes, with the "live" stash containing paths that still have statements to execute and the "deadended" stash containing paths that completed execution. Both stashes were used to generate test cases.

Since the test case generation is very space intensive, symbolic execution was run in a process limited to 40`GB` memory. Symbolic execution continued until it ran out of memory. Generated test cases were saved to files and parsed to be readable by the learning tool.

Improvements to symbolic benchmark execution

Since complexity of library and system calls is a challenge in symbolic execution [5], `angr` has partially implemented the C standard library to reduce state explosion in library calls. The RERS benchmarks heavily rely on taking input from `stdin` by calling the `scanf` function, which is implemented by `angr`. Since the implementation in `angr` has to take into account format strings and large integers, the constraints were more complex than necessary for symbolically executing the benchmarks, because the benchmarks maximally take integers 1 to 20 as their input alphabet. The `scanf` procedure offered by `angr` was modified to attune constraint complexity to the benchmarks by reflecting the reduced input size of the benchmarks.

3.2.3 Test case generation with fuzzing

For the fuzzing group, test cases were generated by Smetsers et al. [34] and made publicly available. The test cases were created using the American Fuzzy Lop tool. Further information on test case generation can be found in the paper by Smetsers et al.

3.2.4 Test case generation with adaptive distinguishing sequences

Test case generation based on adaptive distinguishing sequences (ADS) is currently one of the most efficient ways to check the hypothesis of a model in a black-box setting [25]. Since it is not possible to generate an adaptive distinguishing sequence for all states of all Mealy Machines, test case generation is complemented using the HSI method [30] to be able to generate a complete set of test cases. This complemented method is called hybrid-ADS

For generating ADS test cases a tool to generate testsets developed by J. Moerman was used [28].

3.2.5 Learning setup

To learn Mealy Machines of the benchmarks the state-of-the-art model learning tool LearnLib was used [19]. TTT was used as the learning algorithm [21]. Membership Oracles were implemented using compiled versions of the RERS benchmarks. Equivalence Oracles for symbolic execution and fuzzing were implemented by reading the test cases from files and randomly ordering the test cases to be used. ADS test cases were generated by the tool and were randomly ordered by the tool.

To achieve comparable results between groups, test suite size differences were eliminated by setting a query limit for equivalence checking at 500 queries. Since all testsets are randomly sampled if they're bigger than the limit, the results were averaged over 10 runs to account for randomness.

The main outcome measure of the experiments is the number of found nodes in the models. A higher number of nodes signifies a model that is more closely aligned to the actual behaviour of the program. The time spent learning was also tracked, however the time spent on test case generation was not taken into account. Finally, the size of the queries is also tracked by saving the number of input symbols used for learning.

3.3 Results

The results of the experiments can be found per group in tables 3.1, 3.2 and 3.3. The variance of the number of nodes is generally quite low in the SE and fuzzing groups, except for problems 13, 14 and 15 of the fuzzing group. The reason for this is unclear. The variance in the ADS group fluctuates somewhat; for some problems there is a near-zero variance, for others it is

almost as big as or bigger than the average. This is probably due to the limit on the amount of queries being relatively low for ADS, causing relatively big differences between runs in the number of counterexamples that are found. For example, for problem 7, in four runs five counterexamples were found, whereas three counterexamples were found in the other runs, causing a relatively high variance.

The number of symbols per query is comparable for the SE (weighted average: 6.17 symbols per query) and ADS groups (weighted average: 5.83 symbols per query). The fuzzing group shows a significant difference, averaging 198.42 symbols per query. This indicates that the test cases generated by fuzzing were around 30-fold larger than test cases in the other sets.

The time needed for learning a model was shortest for the ADS group and longest, by a large margin, for the fuzzing group. The difference is most likely attributable to the size of the learned model.

The number of nodes learned per strategy is visualized in Figure 3.1. It clearly shows that the highest number of nodes was found using the fuzzing strategy. Symbolic execution performed second best, finding more nodes than ADS for each problem, but finding as many or fewer nodes than the fuzzing strategy.

RERS Problem	Nodes	Variance	Symbols/Query	Time spent (s)
1	13.00	0.00	8.23	6.95
2	21.90	0.10	7.54	8.35
3	17.40	0.27	7.91	10.95
4	115.80	16.84	5.00	69.57
5	10.40	0.27	8.57	7.92
6	25.70	6.01	8.03	43.08
7	82.30	20.23	5.84	149.61
8	34.60	7.82	7.36	58.43
9	91.40	45.38	5.06	143.11
10	15.00	0.00	8.39	6.18
11	20.44	2.03	6.56	6.84
12	21.80	0.18	5.66	9.24
13	70.00	30.67	4.63	32.96
14	82.33	10.00	4.85	48.29
15	68.50	13.83	4.99	120.80
16	76.20	19.07	4.59	103.54
17	85.50	15.61	3.91	170.67
18	72.50	11.61	5.06	141.35

Table 3.1: **Results of model learning on the RERS 2016 benchmarks using the symbolic execution strategy**
Nodes shows how many nodes were learned on average when constructing a model. *Variance* is the variance of the number of nodes that were learned. *Symbols/Query* is the amount of symbols per query, indicating the size of the queries on average. *Time spent* is the time spent learning the model, single core on an Intel®i5-6600K CPU at 3.5GHz.

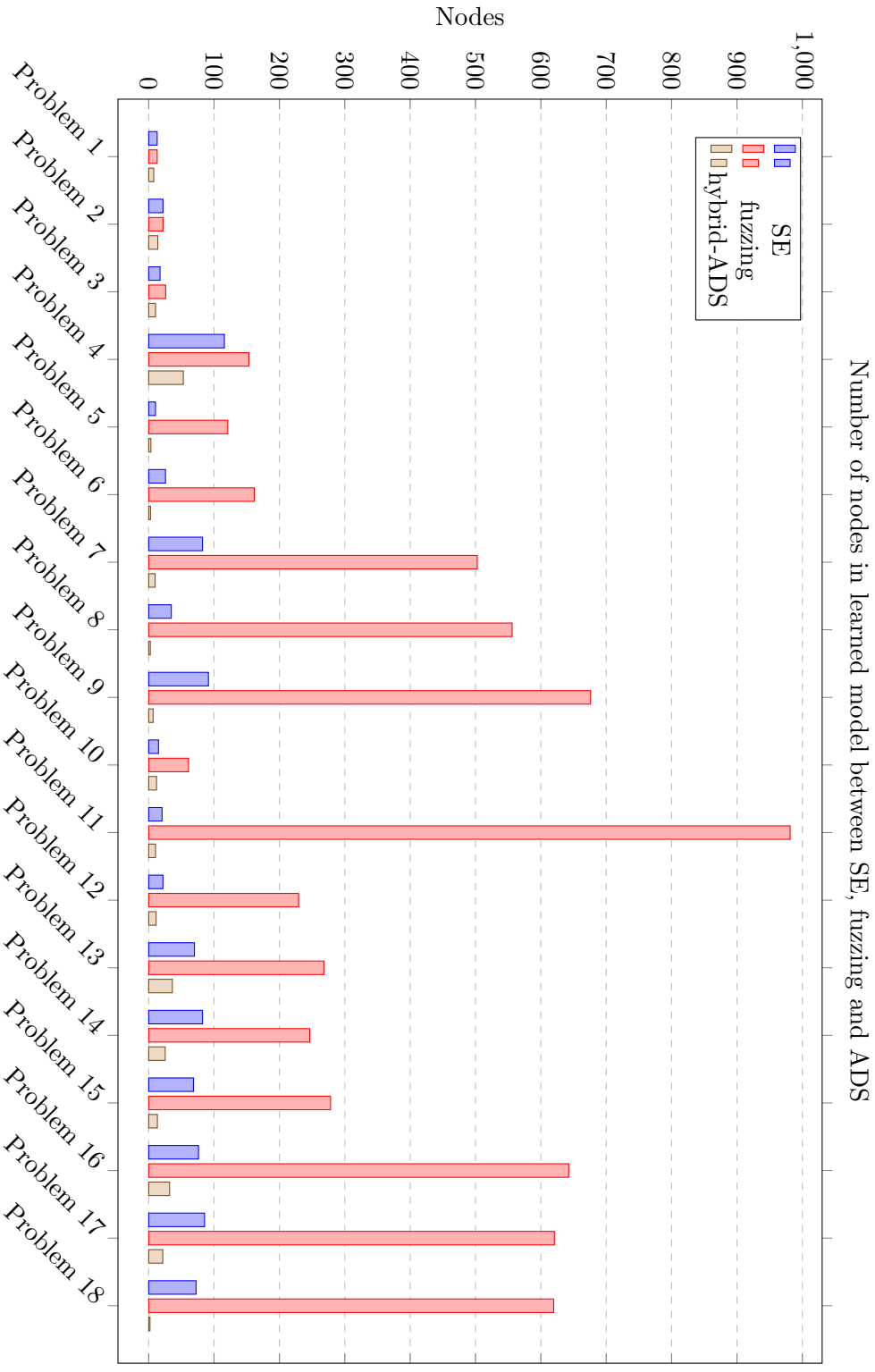
RERS Problem	Nodes	Variance	Symbols/Query	Time spent (s)
1	13.00	0.00	287.58	733.99
2	22.00	0.00	202.82	1,035.57
3	26.00	0.00	294.20	1,466.79
4	153.22	3.19	280.37	3,247.63
5	121.00	0.00	388.62	2,613.37
6	161.50	53.67	242.76	5,917.55
7	502.67	115.87	180.93	7,355.72
8	556.00	0.00	196.68	16,390.96
9	676.00	450.00	135.39	4,948.64
10	61.00	0.00	168.82	269.46
11	981.33	641.33	221.10	13,441.89
12	229.40	537.38	259.59	5,641.46
13	268.00	1,955.43	132.62	1,968.92
14	246.40	1,130.49	137.65	3,629.66
15	278.20	1,730.62	121.05	2,677.48
16	642.57	394.29	103.96	6,228.23
17	620.67	796.33	140.79	19,558.85
18	619.60	248.30	129.83	5,032.79

Table 3.2: **Results of model learning on the RERS 2016 benchmarks using the fuzzing strategy**
Nodes shows how many nodes were learned on average when constructing a model. *Variance* is the variance of the number of nodes that were learned. *Symbols/Query* is the amount of symbols per query, indicating the size of the queries on average. *Time spent* is the time spent learning the model, single core on an Intel®i5-6600K CPU at 3.5GHz.

RERS Problem	Nodes	Variance	Symbols/Query	Time spent (s)
1	7.70	0.23	6.93	5.89
2	13.90	1.21	6.33	6.23
3	10.50	1.83	8.01	8.23
4	53.00	51.78	6.02	28.48
5	3.40	0.93	4.18	3.43
6	2.80	0.18	4.20	3.63
7	9.90	38.77	5.90	12.19
8	2.50	0.28	4.03	3.51
9	6.80	3.73	4.61	5.99
10	12.00	0.00	6.20	3.53
11	10.44	0.28	5.61	4.90
12	11.38	0.27	7.93	6.83
13	36.22	61.69	5.62	14.04
14	25.30	23.34	6.04	12.15
15	13.30	28.23	5.73	12.37
16	32.00	7.71	7.26	37.83
17	21.60	20.71	7.04	24.17
18	2.00	0.00	3.40	2.93

Table 3.3: **Results of model learning on the RERS 2016 benchmarks using the ADS strategy**
Nodes shows how many nodes were learned on average when constructing a model. *Variance* is the variance of the number of nodes that were learned. *Symbols/Query* is the amount of symbols per query, indicating the size of the queries on average. *Time spent* is the time spent learning the model, single core on an Intel®i5-6600K CPU at 3.5GHz.

Figure 3.1:


Nodes
0
100
200
300
400
500
600
700
800
900
1,000
SE
fuzzing
hybrid-ADS
Problem 1
Problem 2
Problem 3
Problem 4
Problem 5
Problem 6
Problem 7
Problem 8
Problem 9
Problem 10
Problem 11
Problem 12
Problem 13
Problem 14
Problem 15
Problem 16
Problem 17
Problem 18
Number of nodes in learned model between SE, fuzzing and ADS

Chapter 4

Discussion

Based on the results described in the previous section, we conclude that symbolic execution can improve the efficiency of conformance testing when compared with the hybrid-ADS black box conformance testing method, but in most cases it lags behind the fuzzing based approach. This indicates that symbolic execution may not be as good an improvement upon conformance checking as previously thought. However, there are some further considerations when interpreting these results:

- The RERS benchmarks may not be representative for many other programs encountered in practice. The RERS problems are designed to be hard, so that software verification and model learning techniques can be compared. However, since many conformance testing methods do not handle large input alphabets well, the biggest alphabet size in the challenge is 20 symbols. In programs that do not take 20 symbols as input, but 16-bit integers for example (having an input alphabet of 2^{16}), symbolic execution may perform much better. Additionally, the way that state is handled in the benchmarks is dissimilar with many programs that are not computer generated, which may be a factor why symbolic execution is performing more poorly than other techniques.

- There are many different symbolic execution tools in use, such as KLEE [7], SAGE [17] or Mayhem [8]. Other symbolic execution engines would surely have led to different results and might have performed better learning the benchmarks. For example, symbolic execution engines like KLEE operate on the source code level. Such tools may be better able to generate interesting test cases, because they have access to higher order information on the workings of programs than binary level symbolic execution engines. Also, proprietary symbolic execution engines may be more powerful than angr, due to greater

funding for development of the tool.

- The test cases in the fuzzing group were generated by running the fuzzing tool for the course of multiple days, while the symbolic execution test cases were generated in approximately 2 hours. Since the Symbolic Exeuction engine used a breadth-first strategy, test cases were generally shorter than the test cases generated by fuzzing. This may cause nodes to be missing when learning with symbolic execution as compared to fuzzing, because the test cases may not be long enough to reach certain states. This problem could be solved by running symbolic execution for a longer time or by employing a different strategy for selecting the execution path to step through.

- Since the test cases of the fuzzing group were a lot longer than test cases in the other groups, setting a limit at 500 queries may have skewed the playing field in favor of the fuzzing group. In future research, it is recommended to limit the total number of symbols used for conformance testing instead. It is hoped that this will create a more level playing field to compare various conformance testing methods.

Although the direct use of symbolic execution does not prove to be better that other examined methods for equivalence checking, it may still improve model learning by automatically creating input or output classes that a mapper can use. Whereas many model learning problems use a small input alphabet (a 20 symbol maximum in case of the RERS challenge), real world applications often range over larger input alphabets. By using the constraints collected by a symbolic execution engine the input space can be partitioned. This potentially greatly decreases the alphabet size for learning setups using a mapper, making for easy learning without the need for human intervention to configure the mapper.

Bibliography

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 10–27, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, Feb 2015.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[5] Roberto Baldoni, Emilio Coppa, Daniele C. D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[6] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] Sang K. Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, May 2012.

[9] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, 1(3):178–187, 1978.

[10] 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*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[12] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286 – 1297, 2010.

[13] Audrey Dutcher. Documentation for the angr suite. `https://github.com/angr/angr-doc`. Accessed: 15-08-2018.

[14] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151, New York, NY, USA, 2017. ACM.

[15] Angelo Gargantini. 4 Conformance Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, pages 87–111, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[16] Maren Geske, Marc Jasper, Bernhard Steffen, Falk Howar, Markus Schordan, and Jaco van de Pol. Rers 2016: Parallel and sequential benchmarks with focus on LTL verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 787–803, Cham, 2016. Springer International Publishing.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[18] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[19] Falk Howar, Malte Isberner, Maik Merten, and Bernhard Steffen. Learnlib tutorial: From finite automata to register interface programs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 587–590, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[20] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018.

[21] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.

[22] Mark Janssen. Combining learning with fuzzing for software deobfuscation. Master's thesis, Delft University of Technology, Delft, NL, 4 2016.

[23] Chris F. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, Dec 1995.

[24] David Lee and Mihalis Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, March 1994.

[25] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[26] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept 1955.

[27] Karl Meinke and Muddassar A. Sindhu. LBTest: A learning-based testing tool for reactive systems. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 447–454, March 2013.

[28] Joshua Moerman. hybrid-ads. `https://gitlab.science.ru.nl/moerman/hybrid-ads`. Accessed: 15-08-2018.

[29] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[30] Alexandre Petrenko and Nina Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, Sept 2005.

[31] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods*, pages 311–325, Cham, 2016. Springer International Publishing.

[32] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[33] Mike Slade. Definitive parallel programming. Master's thesis, University of Warwick, UK, 4 1990.

[34] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing model learning with mutation-based fuzzing. *CoRR*, abs/1611.02429, 2016.

[35] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. The Internet Society, 2009.

[36] Micha Zalewski. American Fuzzy Lop (AFL) fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2015. Accessed: 15-08-2018.