

PARALLELIZING PATH EXPLORATION AND OPTIMIZING CONSTRAINT
SOLVING FOR EFFICIENT SYMBOLIC EXECUTION

by

Junye Wen, M.S.

A dissertation submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
with a Major in Computer Science
December 2020

Committee Members:

Guowei Yang, Chair

Anne Hee Hiong Ngu

Yan Yan

Xiaoyin Wang

COPYRIGHT

by

Junye Wen

2020

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work I, Junye Wen, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

ACKNOWLEDGEMENTS

I am ineffably indebted to my supervisor Dr. Guowei Yang, for his valuable guidance, financial support, encouragement, and patience. This dissertation would not have been possible without his persistent help.

I am extremely thankful and pay my gratitude to the committee members, Dr. Anne Hee Hiong Ngu, Dr. Yan Yan and Dr. Xiaoying Wang, for their constant support and insightful comments in my research.

I extend my gratitude to my classmates and best friends, Mujahid Khan and Tarek Mahmud, for all the inspiration and happiness they gave me in my past years of PhD study.

Grateful acknowledgement is also made to all the faculties and staff in Department of Computer Science at Texas State University, for all the knowledge and help from them and most importantly, for this great PhD program.

Finally, my thanks would go to my beloved family members, for always loving and supporting me unconditionally. Their good examples have taught me and given me confidence to achieve the best I can.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF ALGORITHMS	ix
LIST OF ABBREVIATIONS	x
ABSTRACT	xi
CHAPTER	
I. INTRODUCTION	1
Problem Description	1
Dissertation Topic	2
Organization	4
II. BACKGROUND	6
Symbolic Execution	6
Symbolic Pathfinder	9
Memoized Symbolic Execution	10
Checking Properties with Symbolic Execution	13
Deep Neural Network	14
III. PARALLEL PROPERTY CHECKING WITH STATIC WORKLOAD PARTITIONING	16
Overview	16
STAPAR	19
Evaluation	23
IV. PARALLEL PROPERTY CHECKING WITH STAGED SYMBOLIC EXECUTION	31

Overview	31
STASE	36
Evaluation	40
V. CONSTRAINT SOLVING WITH DEEP LEARNING	51
Overview	51
DeepSolver	53
Symbolic Execution with DeepSolver	59
Evaluation	62
VI. CONCRETIZATION FOR CONSTRAINT ANALYSIS	81
Overview	81
Cocoa	85
Case Study	95
Solving Concretized Constraint in Parallel	101
Evaluation	105
VII. RELATED WORK	111
Parallel Symbolic Execution and Guided Symbolic Execution	111
Machine Learning for Satisfiability Checking	114
Reuse for Efficient Constraint Solving	115
VIII. FUTURE WORK AND DISCUSSION	118
Hybrid of STAPAR and STASE	118
Further Improvement of DeepSolver	118
More Evaluation for Cocoa	119
Testing Deep Neural Networks	119
Balancing the Training Dataset	121
IX. CONCLUSION	124
REFERENCES	127

LIST OF TABLES

Table	Page
1. Results of parallel and regular property checking	26
2. Property checking using guided and prioritized check and regular check.....	29
3. Results of checking properties	45
4. Results of checking properties in WBS	46
5. Results of checking more complicate properties	48
6. Results of checking properties with limited workers.....	50
7. Number of records before and after balancing	67
8. Accuracy of DeepSolver (Balanced via Mutation) in classifying path conditions.....	70
9. Accuracy of DeepSolver (Balanced via GAN) in classifying path conditions.....	71
10. Comparison of Accuracy between Mutation and GAN Balanced Datasets	72
11. Comparison of Loss between Mutation and GAN Balanced Datasets	73
12. Results of symbolic execution with DeepSolver versus GreenTrie.....	77
13. Result of Ranking Heuristics	97
14. Performance of Z3 Solving Simplified PC	100
15. Performance of CVC4 Solving Simplified PC	100
16. Time Cost of Solving PCs from MNIST2	107
17. Time Cost of Solving PCs from MNIST16	108
18. Time Cost of Solving PCs from CIFAR10.....	109

LIST OF FIGURES

Figure	Page
1. Symbolic Execution Sample: (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths	8
2. Illustration of JPF Core	9
3. JPF Listeners	10
4. Example program.....	12
5. (a) Example initial trie. (b) Updated trie in iterative deepening	13
6. Method to compute the middle value of three input numbers and its annotated assertions	18
7. Static analysis approach: An overview of the approach	20
8. Parallel assertion checking: (a) trie explored by the first stage; (b) paths to be explored by multiple workers for parallel assertion checking	34
9. Training DNNs with constraint solutions	53
10. Classifying a path condition using a DNN	58
11. Average time cost (ms) of satisfiability checking.....	75
12. Sample data of MNIST database	82
13. Example Plot of ReLUs	85
14. Example Image from MNIST	95
15. Time Cost Distribution	101
16. Distribution of Time Cost of Solving Concretized PC in Parallel	110

LIST OF ALGORITHMS

Algorithm	Page
1. Algorithm of Guided Property Checking.....	22
2. Algorithm of Finding Feasible Paths to Assertions in the First Stage.....	37
3. Algorithm of Guided Assertion Checking in the Second Stage	39
4. Algorithm of Vectorizing a Canonized PC into a Matrix	56
5. Algorithm of Symbolic Execution with DeepSolver	61
6. Algorithm of Ranking Heuristics.....	92
7. Algorithm of STEP Heuristic.....	94

LIST OF ABBREVIATIONS

Abbreviation	Description
CFG	Control Flow Graph
DNN	Deep Neural Network
GAN	Generative Adversarial Networks
JPf	Java Pathfinder
JVM	Java Virtual Machine
PC	Path Condition
SMT	Satisfiability Modulo Theories
SPf	Symbolic Pathfinder
TACC	Texas Advanced Computing Center

ABSTRACT

Symbolic execution executes programs with symbolic inputs and systematically analyzes program behaviors by exploring all feasible paths. For each path it explores, it builds a path condition, and checks the path’s feasibility by solving its corresponding path condition using off-the-shelf constraint solvers. Symbolic execution is a powerful program analysis technique and has provided a basis for various software testing and verification techniques. However, it remains expensive and is difficult to be applied for large and complex programs due to two major challenges: (1) path explosion problem, i.e., the number of feasible paths in a program grows exponentially with an increase in program size, and (2) constraint solving is expensive.

This dissertation presents four techniques for efficient symbolic execution. The first two techniques, STAPAR and STASE, address the problem of path explosion by parallelizing path exploration in the context of checking properties using symbolic execution. STAPAR statically partitions a check for the whole set of properties into multiple simpler sub-checks, so that different properties are checked in parallel. STASE runs two stages in parallel: one stage for locating all feasible paths to properties and the other stage for checking properties along these paths in parallel. The other two techniques, DeepSolver and Cocoa, optimize constraint solving to improve its efficiency. DeepSolver trains deep neural networks using existing constraint solutions and uses the trained deep neural networks to classify path conditions for their satisfiability. Cocoa reduces the complexity of path conditions by replacing unimportant symbolic variables

with concrete values. Experimental evaluations have shown the efficacy of our techniques compared to the state-of-the-art techniques.

I. INTRODUCTION

Problem Description

Symbolic execution (King, 1976; Clarke, 1976; Godefroid et al., 2005; Sen and Agha, 2006; Păsăreanu and Rungta, 2010; Cadar et al., 2008) uses symbolic values instead of concrete values, as inputs to execute the program, and represents the values of program variables as symbolic expressions on those symbolic inputs. The path condition is a (quantifier free) Boolean formula over the symbolic inputs, collecting constraints on the inputs in order for an execution to follow the associated path. Path conditions are solved by the underlying constraint solver to check the feasibility of the corresponding paths. Thus, symbolic execution can potentially explore all feasible paths of a program. It is a powerful technique for multiple software analyses such as program equivalence checking, regression analysis, and continuous testing (Siegel et al., 2008; Whalen et al., 2010; Yang et al., 2014b). Despite its wide application, symbolic execution still suffers from problems (Anand et al., 2013) that make it difficult to scale to large and complex programs. Specifically, the two major problems are:

- **Path explosion.** Symbolic execution tries to explore all feasible paths of a program to check program behaviors. However, such exhaustive exploration cannot scale to large or complex programs. The number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite for programs with loops. In other words, it is impossible for symbolic execution to exhaustively explore all feasible program paths due to the potentially large number, and only a small subset of paths can be symbolically executed in practice. The goal of discovering a large number of feasible program paths is

further jeopardized because the typical ratio of the number of infeasible paths to the number of feasible paths is high (Ngo and Tan, 2007). This path explosion problem needs to be addressed for efficiency of a symbolic execution system.

- **Expensive constraint solving.** Symbolic execution accumulates branch constraints at conditional statements along a path, and the accumulated constraints, termed path condition, is then solved by an off-the-shelf constraint solver (Barrett and Tinelli, 2007; Cho, 2019; Z3S, 2019) to check the satisfiability of the corresponding path. If a path condition is satisfiable, the corresponding path is feasible and the satisfiable solution to the path condition can be used as an input to execute the program following the corresponding path; otherwise, the path is infeasible and symbolic execution does not continue exploration along the path. However, solving a complex path condition (e.g., path condition involving nonlinear operations such as multiplication and division and mathematical functions such as sin and log) remains a hard problem and can be time consuming. The challenge in solving complex path conditions dampens the efficacy of symbolic execution.

Dissertation Topic

Researchers have developed various approaches to improve the scalability of symbolic execution in the last few years (Yang et al., 2019). For example, some research addresses the path explosion problem by applying parallel algorithm to symbolic execution (Qiu, 2016; Qiu et al., 2017; Kim et al., 2012b), using heuristic-guided path exploration to check interesting paths (Li et al., 2013; Seo and Kim, 2014; Christakis et

al., 2016), or pruning or merging paths and states to reduce the number of paths in the first place (Makhdoom et al., 2014; Yi et al., 2015; Jaffar et al., 2013; Avgerinos et al., 2014). Some other research tries to reduce the cost of constraint solving by applying simplification, reuse and caching mechanism to symbolic execution (Visser et al., 2012; Romano and Engler, 2013; Lloyd and Sherman, 2015) or by leveraging more powerful constraint solvers specially designed for non-linear constraints and bit vectors (Borges et al., 2012; Bagnara et al., 2013; Hadarean et al., 2014). In this dissertation, we introduce multiple techniques to address path explosion and constraint solving problems in symbolic execution. Specifically, we explore two research directions in this work:

- We address the path explosion problem by applying parallelization to symbolic execution in the context of assertion checking using symbolic execution. Our insight is that since assertions are, by definition, side effect free, distribution of assertion checking among multiple workers is a typical embarrassingly parallel problem that is relatively simple and highly effective to apply. Meanwhile, such specific symbolic execution application naturally divides the whole large state space into parts based on whether the state or path is related to assertion checking or not. As a result, we can potentially prune a large portion of the full state space and concentrate computation resources only on the interesting paths to further scale symbolic execution.
- We optimize constraint solving to improve its efficiency. Our first technique in this direction is to apply deep learning and train a model that can quickly classify a path condition regarding its satisfiability. This technique “reuses” the previous

constraint solving results in an innovative way compared to traditional constraint solution reuse techniques. Instead of reusing a constraint solution directly, we use the collective constraint solutions to train a deep neural network as classifier for new path conditions. The second work is to simplify the constraints by replacing a subset of symbolic variables with their concrete input values. With fewer symbolic variables in a path condition, the problem is hopefully simpler and easier to solve by a constraint solver. This idea is inspired by the concept of dynamic symbolic execution, or concolic execution (Godefroid et al., 2005; Sen and Agha, 2006; Cadar et al., 2008; Tillmann and De Halleux, 2008), in which the concrete input is used to help discover new paths to explore in a program. Different from concolic execution, we only replace part of the symbolic variables with concrete values with a purpose to simplify a complex constraint to speed up its solving and thereby speed up symbolic execution as a whole.

Organization

The rest of the dissertation is organized as follows:

Chapter II introduces the background for our research, including symbolic execution, Symbolic Pathfinder, memoized symbolic execution, checking properties with symbolic execution, and deep neural network.

Chapter III presents our first technique, which applies static analysis to distribute assertion checking into sub-checks to be performed in parallel.

Chapter IV presents our second technique, which employs two stages to dynamically partition the state space into several sub-state spaces that can be explored in parallel by symbolic execution.

Chapter V presents our third technique, which speeds up constraint solving in symbolic execution by applying deep learning.

Chapter VI presents our fourth technique, which uses concretization to speed up constraint solving in symbolic execution. It ranks symbolic variables in a path condition according to their importance and replaces the least important ones with their concrete values.

Chapter VII discusses the related work in terms of parallel symbolic execution, guided symbolic execution, machine learning for satisfiability checking, and reuse for efficient constraint solving.

Chapter VIII discusses several directions we would like to explore in future. Chapter IX concludes the dissertation.

II. BACKGROUND

In this chapter, we introduce background for our research, including symbolic execution, memoized symbolic execution, checking properties with symbolic execution, deep neural network, and Symbolic Pathfinder, a widely used symbolic execution framework for Java.

Symbolic Execution

Symbolic execution (King, 1976; Clarke, 1976; Godefroid et al., 2005; Sen and Agha, 2006; Păsăreanu and Rungta, 2010; Cadar et al., 2008) is a powerful analysis technique for systematic exploration of program behaviors, and provides a basis for various software testing and verification techniques, such as program equivalence checking, regression analysis, and continuous testing (Siegel et al., 2008; Whalen et al., 2010; Yang et al., 2014b).

Symbolic execution uses symbolic values, instead of concrete values, as program inputs, and computes values of program variables as symbolic expressions of symbolic inputs. The state of a symbolically executed program includes the values of program variables at a program location, a path condition on the symbolic values to reach that location, and a program counter, which indicates the next statement to be executed. The path condition (Path Condition (PC)) is a Boolean formula over the symbolic input, which is an accumulation of the constraints that must be satisfied by the input for an execution to follow that path.

During symbolic execution, off-the-shelf constraint solvers (Barrett and Tinelli, 2007; Cho, 2019; Z3S, 2019) are used to check the satisfiability of path conditions whenever they are updated. For instance, at each branch point during symbolic execution,

the PC is updated with constraints on the inputs such that (1) if the PC becomes unsatisfiable, the corresponding path is infeasible, and symbolic execution does not continue further along that path and (2) if the PC is satisfiable, a solution to the PC can be used as a program input that executes the corresponding path.

To illustrate, consider the code fragment in Figure 1(a) that swaps the values of integer variables x and y , when the initial value of x is greater than the initial value of y ; we reference statements in the figure by their line numbers. Figure 1(b) shows the symbolic execution tree for the code fragment. A symbolic execution tree is a compact representation of the execution paths followed during the symbolic execution of a program. In the tree, nodes represent program states, and edges represent transitions between states. The numbers shown at the upper right corners of nodes represent values of program counters. Before execution of statement 1, the PC is initialized to true because statement 1 is executed for any program input, and x and y are given symbolic values X and Y , respectively. The PC is updated appropriately after execution of if statements 1 and 5. The table in Figure 1(c) shows the PC's and their solutions (if they exist) that correspond to three program paths through the code fragment. For example, the PC of path (1,2,3,4,5,8) is $X > Y \ \& \ Y - X \leq 0$. Thus, a program input that causes the program to take that path is obtained by solving the PC. One such program input is $X = 2, Y = 1$. For another example, the PC of path (1,2,3,4,5,6) is an unsatisfiable constraint $X > Y \ \& \ Y - X > 0$, which means that there is no program input for which the program will take that (infeasible) path.

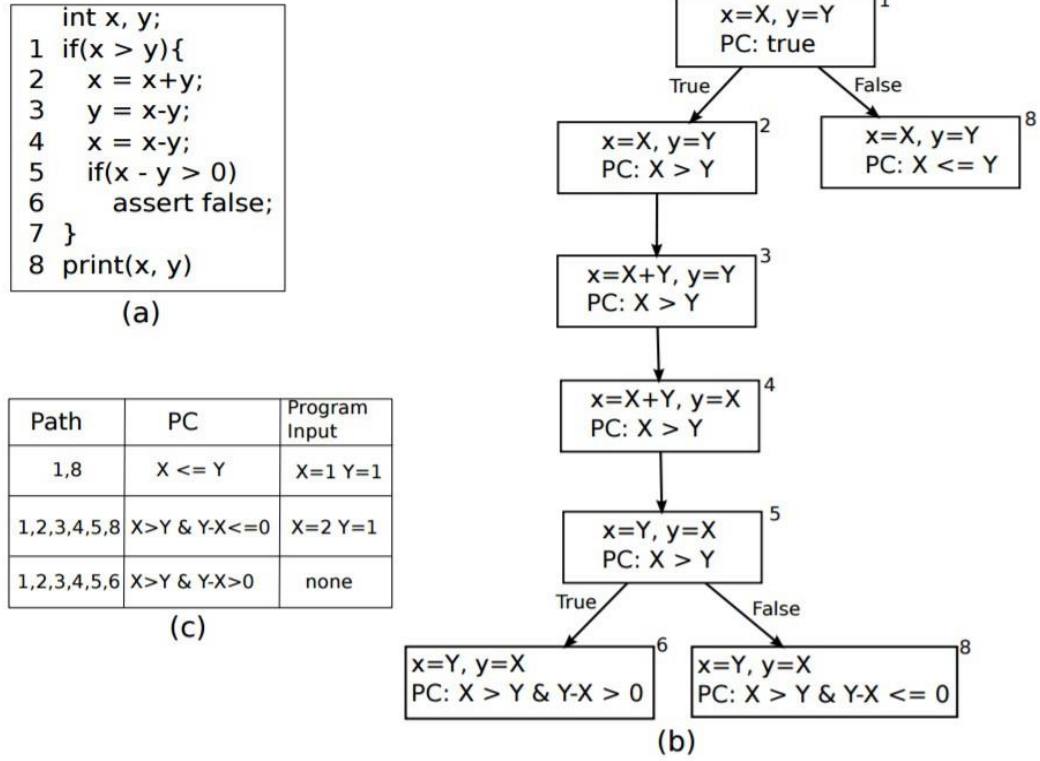


Figure 1: Symbolic Execution Sample: (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

Symbolic execution offers the advantage that one symbolic execution may represent a large, usually infinite, class of normal executions. This can be used to make a great advantage in program analyzing, testing, and debugging. However, to build a symbolic execution system that is both efficient and automatic, three fundamental problems of the technique must be addressed: path explosion, path divergence and complex constraint solving (Anand et al., 2013). In other words, symbolic execution encounters a bottleneck that a target subject becomes too large and complicated to be fully explored in a reasonable time. In our research, we introduce different techniques to handle such difficulties in multiple aspects.

Symbolic Pathfinder

Symbolic Pathfinder (SPF) is a Java framework specially designed for symbolic execution. SPF combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs instead of concrete values. Values of variables are represented as constraints over the symbolic variables, generated from analysis of the code structure. These constraints are then solved to generate test inputs. Essentially SPF performs symbolic execution for Java programs at the bytecode level. SPF uses the analysis engine of Java Pathfinder (JPF), a model checker for Java. The core of JPF (Havelund and Pressburger, 2000) is a modified virtual machine for Java bytecode which is generally used as model checker. A brief illustration of its workflow is shown in Figure 2 (Picture taken from previous JPF wiki: <https://github.com/javapathfinder/jpf-core/wiki>). It is used to find defects in given Java programs, with the assertions, or specifications, to check given as input. JPF gets back to user with a report that says if the assertions hold and/or which verification artifacts have been created by JPF for further analysis (like test cases).

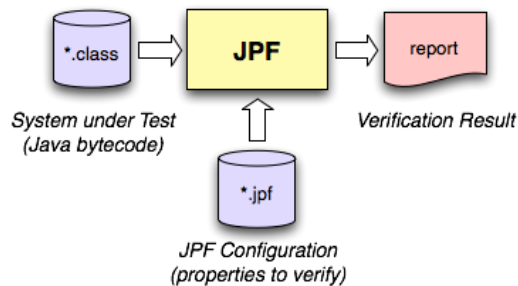


Figure 2: Illustration of JPF Core

The Java Virtual Machine (JVM) of JPF is the Java specific state generator. By executing Java bytecode instructions, the JVM generates state representations that can be

checked, queried, stored, and restored. JPF provided interfaces for developer to control these behaviors. As shown in Figure 3 (Picture taken from previous JPF wiki: <https://github.com/javapathfinder/jpf-core/wiki>), listeners are used to monitor certain events (e.g. instruction executed, choice generator registered, state backtracked, etc.). By implementing listeners, developers can control the flow of execution or gather important information in the process.

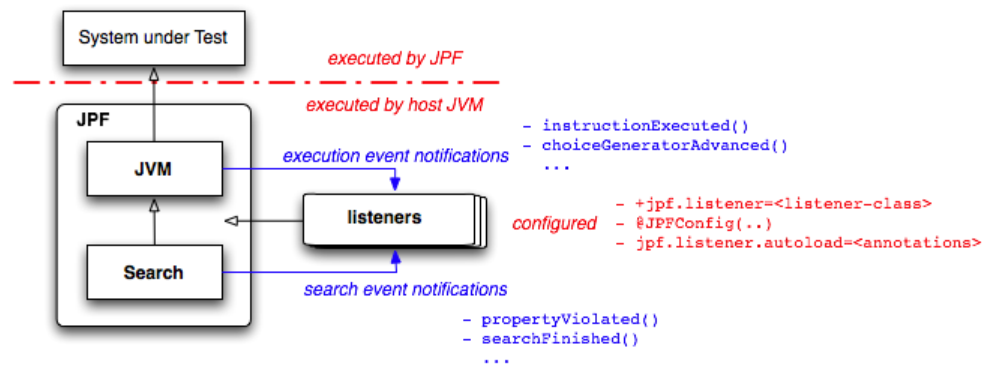


Figure 3: JPF Listeners

SPF is an extension of JPF. It combines model checking of JPF with symbolic execution features. Model checking is used to explore different symbolic program executions, to systematically handle aliasing in the input data structures, and to analyze the multithreading present in the code, while the extended behavior introduced as listeners execute the program on symbolic inputs instead of concrete values and uses mounted solvers to solve the constraints and automatically generate test cases or determine path feasibility. It is a widely applied tool in academia and industry.

Memoized Symbolic Execution

The key insight in memoized symbolic execution (Memoise) is that applying symbolic execution often requires several successive runs of the technique on largely

similar underlying problem instances. Memoise leverages the similarities to reduce the total cost of applying the technique by maintaining and updating the computations involved in a symbolic execution run. It reduces both the number of paths to explore by pruning the path exploration as well as the cost of constraint solving by re-using previously computed constraint solving results.

Specifically, Memoise uses a trie (Fredkin, 1960; Willard, 1984), an efficient tree-based data structure, for a compact representation of the paths visited during a symbolic execution run. Essentially, the trie records the choices taken when exploring different paths, together with bookkeeping information that maps each trie node to the corresponding condition in the code. Maintenance of the trie during successive runs allows re-use of previous computation results of symbolic execution without the need for re-computing as is traditionally done. Constraint solving is turned off for previously explored paths and the search is guided by the choices recorded in the trie. An initial run of Memoise performs standard symbolic execution as well as builds the trie on-the-fly and saves it on the disk for future re-use. To facilitate future runs of symbolic execution, a subset of the leaf nodes in a trie is partitioned into a set of boundary nodes, which are leaf nodes because of the chosen depth bound, and a set of unsatisfiable nodes, which are leaf nodes due to unsatisfiable path conditions. Based on the results cached in the trie, Memoise enables efficient re-execution, which is guided by the trie using algorithms that are specialized for the particular analysis that is performed.

```

int m(int curr, int thresh, int step){
    int delta = 0;
    if (curr < thresh){ //bytecode #5
        delta = thresh - curr;
        if ((curr + step) < thresh) //bytecode #17
            return -delta;
        else
            return 0;
    } else {
        int counter = 0;
        while (curr >= thresh) { //bytecode #31
            curr = curr - step;
            counter++;
        }
        return counter;
    }
}

```

Figure 4: Example program.

We illustrate Memoise for iterative deepening on the program in Figure 4. Method *m* has three integer inputs: *curr* (current), *thresh* (threshold) and *step*, and calculates the relationship between the current and the threshold, in increments given by the step value. Since symbolic execution of programs with loops may result in an infinite number of paths to explore, symbolic execution is often run using iterative deepening, where a depth bound of the search is set for symbolic paths, and is iteratively increased until either an error is found or the desired testing coverage has been achieved. Memoise enables an efficient iterative deepening, by only considering paths bounded by the search depth bound to be re-executed during the new iteration, since other paths are ended naturally by execution at smaller depths and hence cannot have deeper successors. The paths that lead to boundary nodes are thus selected and guided by the trie, are executed up to the next depth bound. Figure 5 (a) shows the trie for a symbolic execution run on the program bounded at depth 4. The two nodes (5,1 - 31,0 - 31,0) and (5,1 - 31,0 - 31,1)

are boundary nodes and the node (5,1- 31,1) is an unsatisfiable node. During re-execution constraint solving is turned off for the portion of the path that has already been explored in the previous iteration. Figure 5 (b) shows the updated trie bounded at depth 5, in which only paths highlighted (following 5,1 - 31,0) are explored. Memoise requires only 2 calls to the constraint solver to generate this trie; in contrast, standard symbolic execution would require 10 calls (1 for each branch).

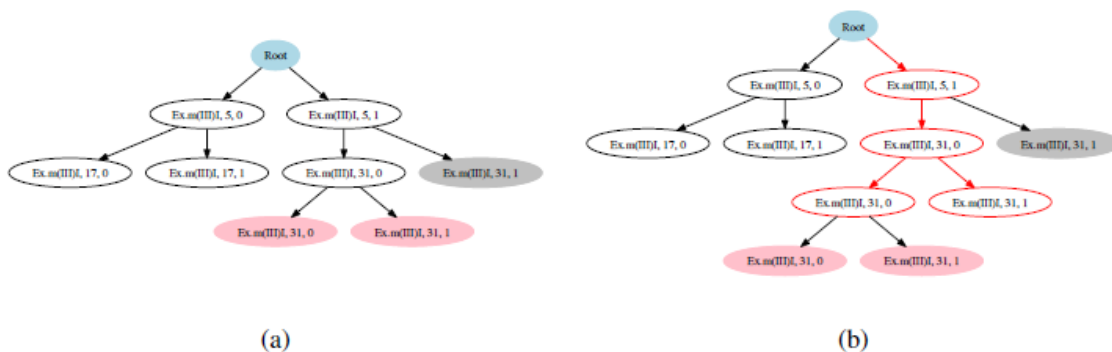


Figure 5: (a) Example initial trie. (b) Updated trie in iterative deepening.

Checking Properties with Symbolic Execution

Annotating functional correctness properties of code, e.g., using assertions (Clarke and Rosenblum, 2006) or executable contracts (Leavens et al., 2005; Meyer et al., 1987), enables automated conformance checking of program behaviors to expected properties, and is widely used for finding bugs (Corbett et al., 2000; Godefroid, 1997). However, effectively utilizing such properties in practice is complicated, in part due to the high computational cost of checking them.

When programs are annotated with functional correctness properties, symbolic execution can be naturally applied to automatically check program behaviors against the annotated properties to check their validity. Symbolic execution can systematically explore the program's state space to find paths to assertion violations and provide users

with a counterexample for each violation by solving the corresponding path condition using the underlying constraint solver. However, suffering from problems of path explosion and complicated constraint solving, checking assertions using symbolic execution can be expensive. Symbolic execution is usually configured either to explore all the state space to find all possible assertion violations, or to stop when it finds the first assertion violation. In the former case, one symbolic execution run could take a long time before giving any results, and users have to wait for the whole exploration to complete before they could take any action to deal with the potential problems in their code or assertions. In the latter case, users have to run symbolic execution for multiple times to find all assertion violations in case there exists multiple assertion violations.

Deep Neural Network

Deep Neural Network (DNN) have been widely used in many artificial intelligence areas, such as computer vision (Krizhevsky et al., 2017), natural language processing (Socher et al., 2013), and speech recognition (Mohamed et al., 2012). In a deep learning model, multiple layers of information processing stages in hierarchical architectures are utilized for pattern classifications or feature learning purposes. A typical DNN has one input layer which takes in input data, one output layer which generates final classification results, and several hidden layers to perform intermediate processing (e.g., feature extraction). Each layer of a DNN is comprised of nodes, termed neurons. The neurons refine and extract information based on the values received from the previous layer, and then compute a value for the next layer. DNNs use the multiple hidden layers to progressively extract higher level features from the input dataset.

DNNs are commonly used as *classifiers*. Each record of an input dataset contains a pre-set label or class for classification, while each neuron in the output layer represents one such class. Each neuron takes inputs from other neurons (typically from the ones in the previous layer) and computes an output by applying an activation function (e.g., ReLUs or Sigmoid) to the weighted sum of its inputs according to a unique weight vector and a bias value. The output is then sent to other neurons for computation and propagation until it reaches neurons in the output layer. The computation result of an output neuron represents the “confidence” or the “possibility” of the input record to fall into the corresponding class, which is used by the DNN to make final decisions of the classification based on user’s design (e.g. outputting one or more classes with highest possibility, or outputting only classes with confidence scores over a certain threshold (Nielsen, 2018). In our work, we use rectified linear units (ReLUs) (Nair and Hinton, 2010) as activation function.

III. PARALLEL PROPERTY CHECKING WITH STATIC WORKLOAD PARTITIONING

This chapter presents STAPAR, a technique to improve the scalability of symbolic execution on checking properties in parallel with a static analysis approach. Our approach partitions a check for the whole set of properties into multiple simpler sub-checks—each sub-check focusing on a single property, so that different properties are checked in parallel among multiple workers. Furthermore, each sub-check is guided by the checked property to avoid exploring irrelevant paths and is prioritized based on distances towards the checked property to provide early feedback. We implement our approach in SPF and experiments on systematically checking assertions in Java programs show the effectiveness of our approach.

This chapter is based on our previous publication in 2018 (Wen and Yang, 2018).

Overview

Advances in symbolic execution have been made during the last decade. Specifically, parallel analysis (Bucur et al., 2011; Staats and Păsăreanu, 2010; Siddiqui and Khurshid, 2010, 2012) allows multiple workers to explore largely disjoint sets of program behaviors in parallel, and has shown promise in addressing the scalability issue of symbolic execution. However, to the best of our knowledge, none of the approaches consider the characteristics of the annotated properties in their parallelization strategies. Our key insight of this approach is that properties are normally written without side effects, and thus checking of each property is independent of checking of other properties. Our approach partitions a check for the whole set of properties into multiple simpler sub-checks—each focusing on one single property, so that different properties are

checked in parallel among multiple workers. Furthermore, each sub-check is guided by the checked property to avoid exploring irrelevant paths and is prioritized based on distances towards the checked property to provide earlier feedback, allowing users to fix bugs in code or refine properties earlier. Specifically, during state space exploration we statically check whether the checked property is reachable or not along the current path and prune the search when the checked property cannot be reached. Moreover, we prioritize the state space exploration so that the state whose corresponding location has the shortest distance towards the checked property is explored first, i.e., the shortest path to the checked property gets explored first. Therefore, the prioritized state space exploration can provide earlier feedback on the checked property. Note that the chance of pruning irrelevant state space is much higher in each sub-check than in the original check, since in a sub-check the program under analysis has only one property at a particular location in the program, while the program under analysis in the original check has multiple properties scattered in different locations in the program.

We implemented our approach in SPF (Păsăreanu et al., 2013). To evaluate the efficacy of our approach we apply it in the context of symbolic execution for checking Java programs annotated with assertions. We conduct experiments based on five subjects: three Java programs with manually written assertions and two Java programs with synthesized assertions. Experimental results show that our approach for parallel property checking detects more assertion violations and reduces the overall analysis time compared with regular non-parallel property checking. For one subject, while regular property checking timed out after executing for two hours, our parallel property checking technique completed within four seconds. In addition, for most sub-checks, our guided

check prunes state space and reduces the time cost, and our prioritized check provides earlier feedback compared to regular check.

We use an example to illustrate how our approach leverages the annotated properties to improve the scalability of symbolic execution for property checking.

Consider the source code shown in Figure 6. It computes the middle value of its three integer inputs; this method is adapted from previous work of Jones (2008), and five assertions are manually added to check the correctness of the program. For example, the user asserts $x \leq y \ \&\& \ y \leq z$ at line 4, indicating that y should be the middle value of the three inputs; otherwise, an assertion violation is captured.

```
int median(int x, int y, int z) {
    if (y < z) {
        if (x < y){
            assert x <= y && y <= z; // #1
            return y;}
        else if (x < z){
            assert y <= x && x <= z; // #2
            return x;}
    }
    else {
        if (x > y){
            assert z <= y && y <= x; // #3
            return y;}
        else if (x > z){
            assert z <= x && x <= y; // #4
            return x;}
    }

    assert (x<=z && z<=y) || (y<=z && z<=x); // #5
    return z;
}
```

Figure 6: Method to compute the middle value of three input numbers and its annotated assertions.

The workload of checking five assertions in this program is conducted by five workers running in parallel, such that each worker checks one single assertion. For example, the worker responsible for checking assertion #1 analyzes a program version, where the code together with the target assertion #1 remain unchanged, while all the other four assertions are removed.

In addition, each sub-check is further optimized using guided and prioritized state space exploration based on the checked assertion. For checking assertion #1, the sub-check is guided by assertion #1, avoiding exploring the irrelevant parts of the program. Therefore, instead of exploring all the six possible paths in the program, the guided check only explores one path, that satisfies path condition $y < z \ \&\& \ x < y$ and reaches the checked assertion. It results into up to 5/6 reduction in terms of the number of paths to be explored. If multiple paths can reach the checked assertion, we use shortest distance-based heuristics to prioritize the search so that the assertion can be checked as early as possible and a feedback, i.e., whether the assertion is violated or not, can be returned to the user as early as possible.

STAPAR

STAPAR is focused on how to optimally utilize the computing resources available to check properties, specifically in a parallel setting where the checking can be conducted among several workers. Our key insight is that properties are normally written without side effects, and thus checking of each property is independent of checking of other properties. The result from checking all properties in one run should be the same as that from checking properties in multiple runs in parallel. This enables us to partition a check for the whole set of properties into multiple simpler checks—each focusing on one

single property, so that different properties are checked in parallel among multiple workers. Therefore, the original check is converted into multiple simpler sub-checks in parallel for better scalability.

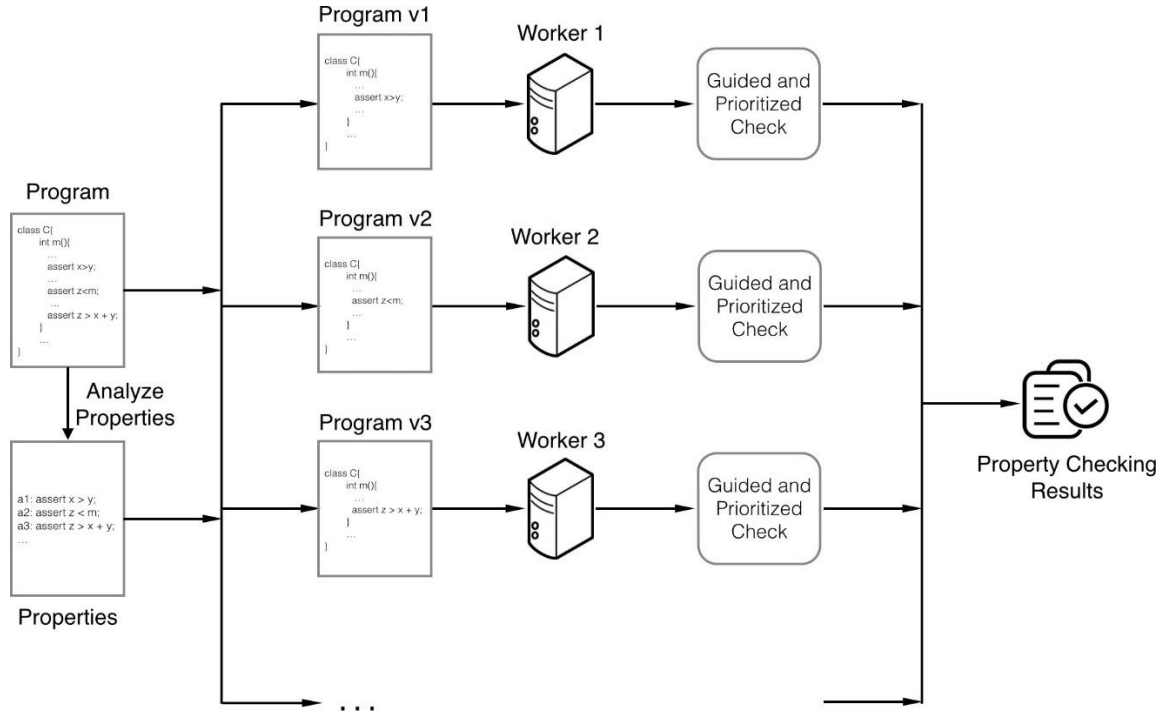


Figure 7: Static analysis approach: An overview of the approach

Figure 7 shows an overview of the approach. Consider a program P with multiple properties $PT = \{PT_1, PT_2, \dots, PT_m\}$ to check. Our approach first statically analyzes the program to find all the m properties to check, and accordingly prepare m program versions $V = \{v_1, v_2, \dots, v_m\}$ where each version contains only one property that does not appear in other versions. These versions are then checked by m workers, each worker focusing on one version and altogether checking all the properties in parallel. Each worker works on its own program version with one single property using guided and

prioritized check. Finally, the property checking results from these workers are delivered to the user.

The partition of properties not only simplifies the program to be checked due to the removal of other properties, but also allows further optimization of each sub-check. Since each sub-check focuses on one single property, it is more likely to have paths that do not reach the checked property compared with the original check that focuses on multiple properties. Leveraging this observation, each sub-check, i.e., a symbolic execution run for checking one single property, is guided by the checked property such that it only explores the program state space that is relevant to the checked property. If the current path cannot reach the checked property, symbolic execution does not continue along the path and backtracks. By effectively pruning paths that cannot reach the checked assertions, our approach avoids the cost of exploring irrelevant paths.

Algorithm 1 shows the procedure check for performing property checking for a program with one single property. Given as input a program, a property to check, and a bound on the search depth, the procedure checks the conformance of the program behaviors with the checked property and return all property violations in the program. It starts with the initial state for s , 0 for $depth$, and an empty set for VS . It finds all enabled transitions at the current state (Line 1) to systematically search the state space. Lines 4-5 locate the Control Flow Graph (CFG) nodes for the enabled transition, and the checked property, respectively. Both the enabled transition and the checked property could correspond to multiple CFG nodes, we simplify it here assuming that each corresponds to one CFG node. It checks whether the current transition reaches the checked property, and if not prune the search (Lines 6-7); otherwise, it executes the transition to get to the next

state and update the pc and depth (Lines 9-11). If pc is unsatisfiable (i.e., the corresponding path is infeasible), the checked property is violated, or search depth reaches the bound, it backtracks to explore other un-explored enabled transitions (Lines 12-20); otherwise, it recursively explores the states rooted at the new state s' (Line 22).

Algorithm 1 Algorithm of Guided Property Checking

Input: Program P , property PT , search depth bound $DepthBound$

Output: A set of property violations detected during symbolic execution VS

```

1: Queue  $tq \leftarrow$  enabled transitions at current state  $s$ 
2: while  $tq \leftarrow isEmpty()$  do
3:    $t \leftarrow tq.remove()$ 
4:    $nt \leftarrow GetCFGNode(P, t)$ 
5:    $na \leftarrow GetCFGNode(P, PT)$ 
6:   if  $\neg IsCFGPath(nt; na)$  then
7:     continue
8:   else
9:      $s' \leftarrow execute(s, t)$ 
10:     $pc \leftarrow$  current path condition
11:     $depth \leftarrow depth + 1$ 
12:    if  $pc$  is not satisfiable then
13:      continue
14:    end if
15:    if  $isPropertyViolated(s')$  then
16:       $VS.add(violation(s'))$ 
17:      continue
18:    end if
19:    if  $depth = DepthBound$  then
20:      continue
21:    else
22:       $check(s')$ 
23:    end if
24:  end if
25: end while

```

In addition, each sub-check is prioritized to provide early feedback to the user. In the context of property checking, usually one property violation is enough for investigating the violation, and there is no need to find all property violations. Our insight is that the earlier a property is checked, the earlier the user could start the investigation

and fix the potential problem either by modifying the code or by refining the checked property. As there is no precise way to predict the feasibility of paths and how long each path would take. We use a heuristic to prioritize the check. Specifically, we calculate the distances from the current point towards the checked property along all potential paths and choose the shortest path to explore first (Ma et al., 2011).

To prioritize the search, at each branching point, we sort the list of enabled transitions based on an estimated distance to the checked property in a CFG. For each enabled transition ti , we compute an estimated distance to the checked property. The enabled transitions queue (tq in Algorithm 1) is sorted in ascending order based on the estimated distances of the transitions before the queue is explored. The enabled transition with the shortest distance is explored first. The distance is a lower bound on the number of CFG branches from a node ni (corresponding to ti) to node nj , that is corresponding to the checked property: $\forall ni . nj : di := \min (branches (ni, nj))$.

In our approach, we use the all-pairs shortest path algorithm to compute the lower bound on the number of CFG branches. The complexity is cubic in the number of branches in the CFG. We note that metrics other than number of branches can also be used as a distance estimate, for example, the number of bytecodes.

Evaluation

We name our approach STAPAR, which stands for static partitioning. We empirically evaluate the effectiveness of STAPAR.

Our evaluation addresses the following research questions:

- **RQ1:** How does the efficiency of STAPAR compare with regular property checking?

- **RQ2:** How does the cost of guided check compare with regular check?
- **RQ3:** How does prioritized check compare with regular check in terms of providing feedback to the user?

In our evaluation, we use five subjects including median, testLoop, trityp, Wheel Brake System (WBS), and Traffic Collision Avoidance System (TCAS). All of them have been used before for evaluating symbolic execution techniques (Staats and Păsăreanu, 2010; Yang et al., 2014b, 2012, 2014a).

The first subject median is shown in Figure 6.

The second subject testLoop is used to investigate STAPAR can help deal with loops, as they pose particular challenges to symbolic execution and handling them efficiently is an active area of research.

The third subject is a Java version of the classic triangle classification program by Ammann and Offutt. The classification logic of the trityp program. We consider the correct version of assertions developed for trityp in previous work (Yang et al., 2014a).

For the two subject programs WBS and TCAS, we use mechanically synthesized assertions. To synthesize assertions for our experiments, we use the Daikon tool for invariant discovery (Ernst, 2000). Specifically, we apply Daikon on each subject to discover invariants and transform them to assertions. Daikon requires a test suite to execute the program under analysis and detect its likely invariants. TCAS had a test suite available in the Software Infrastructure Repository (SIR), so we used this test suite which contains 1608 tests. For WBS, we wrote a random test generator to create a test suite with 1000 tests. We selected all the eight Daikon invariants for TCAS and randomly selected 25 out of 35 invariants for synthesizing assertions.

Experiment Setup

In this work, we use Symbolic PathFinder (SPF) (Păsăreanu et al., 2013), an open-source tool for symbolic execution of Java programs built on top of the Java PathFinder (JPF) model checker (Visser et al., 2003) to perform symbolic execution. We implemented guided and prioritized check in SPF as a customized listener, and we built customized control flow graphs to compute estimated distances and reachability information to guide and prioritize property checking. We also conduct experiments using regular symbolic execution as implemented in SPF for comparison. Choco constraint solver (Cho, 2019) is used for solving path conditions involved in symbolic execution.

To evaluate RQ1 and RQ2, symbolic execution is configured to detect all assertion violations; while to evaluate RQ3, symbolic execution is configured to stop when it detects the first assertion violation, to check whether our prioritized check could provide earlier feedback than regular check.

We perform the experiments on the Lonestar cluster at the Texas Advanced Computing Center (TACC) (lon). TACC provides powerful computation nodes with reliable and fast connectivity. The programs for each worker node are executed on independent processors without memory sharing.

We assume that there are enough workers available for performing the tasks in parallel. We consider it is reasonable to make this assumption, for the workload in STAPAR is distributed based on the number of assertions, which is relatively small. The largest number of assertions in subjects is TCAS, which has 25 sub-versions in total, while a cluster for parallel processing can easily handle the task with no problem. For

instance, the Lonestar cluster we use to run the experiments has 64 nodes in a chassis, with 24 cores per node. The computing resource is, with no doubt, enough to cover our tasks for evaluation.

Results and Analysis

In this section, we present the results of our experiments, and analyze the results with respect to our three research questions.

Table 1: Results of parallel and regular property checking.

Subject	STAPAR				Regular Property Checking			
	Violations	Time (s)	# of States	Memory (MB)	Violations	Time (s)	# of States	Memory (MB)
Median (5 assertions)	0	2 (0-2)	5 - 13	965 - 965	0	2	13	965
testLoop (2 assertions)	2	31 (0-30)	103 - 180	965 - 965	-	TO	-	-
trityp (10 assertions)	0	49 (18-48)	33 - 49	965 - 965	0	103	81	965
WBS (8 assertions)	222	7 (0-7)	359 - 671	965 - 1,178	92	2	533	965
TCAS (25 assertions)	251	680 (27-679)	679 - 935	965 - 1,685	195	2,025	2,047	965

- *RQ1: How does the efficiency of STAPAR compare with regular property checking?*

Table 1 shows the experimental results for checking all assertions in the subject programs using STAPAR and using regular non-parallel property checking approach. It shows the number of detected assertion violations, and three types of checking cost, i.e., time, number of states explored, and the maximum memory cost, for each approach.

Since in the parallel property checking sub-checks are analyzed in parallel among multiple workers, the table shows cost ranges of values across all sub-checks, and it also shows the overall time cost for the parallel property checking; while for regular symbolic execution the cost is collected by running regular symbolic execution on the original program annotated with all assertions. We note that 0 in time cost means less than 1 second. TO indicates that the corresponding check timed out. We find that there are no assertion violations for median and trityp, while for the other three subjects, the parallel approach detects more assertion violations than regular approach. This is because some expensive assertion checking happens only in the parallel property checking. Since Symbolic PathFinder backtracks as soon as it detects an assertion violation, the inputs reaching deep assertions may be reduced due to violations of the shallow assertions along the same path, and thus may not detect the possible violations of the deep assertions in regular property checking approach. Moreover, for all subjects except for WBS, the parallel approach is more efficient than regular approach in property checking. Specifically, it achieves almost 3X speedup for TCAS. For testLoop, while regular symbolic execution timed out after executing for two hours, the parallel property checking completed within 31 seconds. Without surprise, most sub-checks explored only part of the state space. We also note however for WBS, STAPAR took more time, and explored more states, which is because of the cost for detecting the 130 more violations.

In addition, we find that although the parallel approach takes almost the same memory cost as regular symbolic execution for most runs, it takes more memory for some sub-checks for WBS and TCAS; we note however that the maximum memory

reported by SPF may vary a lot due to the underlying garbage collection, and thus this comparison is not very meaningful.

- *RQ2: How does the cost of guided check compare with regular check?*

Table 2 reports the experimental results for each sub-check using guided check and prioritized check compared to using regular check, i.e., regular symbolic execution. As we explained before, the comparison in memory cost is not very meaningful, thus here we only report the cost in terms of time and explored states.

To evaluate RQ2, symbolic execution is configured to check for all assertion violations. We observe that for 44 out of 50 versions, the guided check explored fewer states than the regular check, since the guided check prunes state space exploration when the checked property is not reachable. For example, for v1 of testLoop, guided check explored 103 states while regular check explored 154 states, which is about 1/3 reduction. Accordingly, the guided check took less time than regular check for most of these cases. For example, for v1 of trityp, guided check took 18 seconds while regular check took 22 seconds. However, we note that for some cases, although there was a reduction in states, the time cost of guided check was even higher than regular check due to the overhead of static analysis involved in guided check.

Table 2: Property checking using guided and prioritized check and regular check.

Subject	Ver	Check all violations				Check first violation			
		Guided Check		Regular Check		Prioritized Check		Regular Check	
		Time	States	Time	States	Time	States	Time	States
median	v1	0	5	0	11	0	5	0	11
	v2	0	7	1	11	0	7	1	11
	v3	0	5	1	11	0	5	1	11
	v4	1	5	0	11	0	5	1	11
	v5	2	13	2	13	1	13	1	13
testLoop	v1	0	103	0	154	0	103	0	154
	v2	30	727	TO	TO	0	180	TO	TO
trityp	v1	18	33	22	40	18	33	22	40
	v2	18	35	18	36	18	35	21	36
	v3	33	49	40	57	34	49	35	57
	v4	29	39	29	39	29	39	32	39
	v5	48	35	53	36	45	35	55	36
	v6	28	37	30	42	28	37	29	42
	v7	26	37	27	40	26	37	29	40
	v8	19	35	20	36	21	35	21	36
	v9	22	39	19	42	22	39	20	39
	v10	20	39	23	39	23	39	21	39
WBS	v1	0	451	0	455	0	163	0	255
	v2	0	359	0	359	0	222	0	341
	v3	0	527	1	530	0	527	0	530
	v4	0	623	1	623	0	9	0	9
	v5	0	535	1	535	0	535	0	561
	v6	7	671	11	680	0	9	0	9
	v7	0	487	1	500	0	48	0	117
	v8	0	527	0	530	0	368	0	421
TCAS	v1	219	727	250	760	289	727	265	702
	v2	27	727	27	760	37	727	43	702
	v3	34	687	33	702	37	687	33	702
	v4	149	687	156	702	125	687	137	702
	v5	30	679	41	754	27	679	34	754
	v6	35	679	40	754	33	679	37	754
	v7	31	679	35	679	33	679	34	679
	v8	28	679	33	679	33	679	34	679
	v9	241	695	275	722	240	695	257	722
	v10	251	695	270	722	222	695	318	722
	v11	32	695	36	722	1	33	1	38
	v12	31	695	33	722	1	33	1	38
	v13	201	695	226	727	238	695	241	727
	v14	130	695	132	727	134	695	146	727
	v15	28	695	34	727	11	229	13	270
	v16	28	695	35	727	9	229	12	270
	v17	679	743	644	745	557	743	568	745
	v18	31	743	32	745	31	743	32	745
	v19	36	935	34	950	15	370	26	439
	v20	33	935	36	950	8	247	14	323
	v21	30	719	30	874	29	678	29	678
	v22	34	719	35	874	11	167	18	214
	v23	33	815	35	827	0	20	0	33
	v24	28	815	39	827	10	191	20	331
	v25	34	815	35	827	9	211	19	231

- *RQ3: How does prioritized check compare with regular check in terms of providing feedback to the user?*

To evaluate RQ3, symbolic execution is configured to stop when it finds the first assertion violation. From Table 2, we observe that for 40 out of 50 versions, prioritized check explored fewer states than regular check, and for 8 versions, both techniques explored the same number of states. For instance, for v24 of TCAS, prioritized check explored 191 states, while regular check explored 331 states. However, for the other 2 versions (i.e., v1 and v2 of TCAS, prioritized check explored slightly more states than regular check. This is not surprising as the shortest path selected by our heuristics is based on number of branches in CFG and may result in more states to explore in symbolic execution. Similar to previous experiments, prioritized check usually took less time when it explored fewer states, as the time cost is correlated with states exploration. For example, for v10 of TCAS, prioritized check took 222 seconds, while regular check took 318 seconds, which is about 1.5X speedup. Moreover, for v2 of testLoop, prioritized check completes in less than one second; in contrast, regular check timed out after running for two hours. Only for few versions, prioritized check took slightly more time than regular check.

IV. PARALLEL PROPERTY CHECKING WITH STAGED SYMBOLIC EXECUTION

This chapter presents STASE, a technique to improve the scalability of symbolic execution on checking properties in parallel with a dynamic analysis approach. We introduce the approach for checking properties in parallel with staged symbolic execution. It consists of two stages running in parallel, one stage for finding all feasible paths to properties and the other stage for checking properties along these paths in parallel. Although there is some redundancy in state space exploration during the latter stage, we leverage memoization to efficiently explore the parts that have been explored before. We implement our approach on top of Symbolic PathFinder and evaluate it on several Java subjects with assertions. The experimental results show the effectiveness of our approach compared to sequential property checking using conventional symbolic execution. In particular, our approach finds the same assertion violations as sequential property checking while achieving up to 2.97X speedup for checking assertions and up to 5.65X speedup for checking more complex assertions, thereby our approach provides users earlier reports of assertion checking.

This chapter is based on our publication in 2019 (Wen and Yang, 2019).

Overview

We introduce a novel approach for parallel property checking with staged symbolic execution to improve the efficiency of symbolic execution on assertion checking. It consists of two stages running in parallel: the first stage focuses on finding all feasible paths to assertions as target paths, and the second stage checks assertions on each target path in parallel. Although there is some redundancy of state space exploration

during the second stage, we leverage memoization (Yang et al., 2012) to efficiently explore the parts that have been explored before.

Our insight is to distribute the work of assertion checking to several workers in parallel, thus each worker solves a much simpler problem than exploring the whole state space. As a rule, assertions are usually written without any side effect, therefore the checking of one assertion is independent of the checking of other assertions. Thus, it guarantees that distribution of assertion checking would not change the final result. Compared with existing parallel symbolic execution techniques (Staats and Păsăreanu, 2010; Siddiqui and Khurshid, 2010; Kim et al., 2012a; Bucur et al., 2011; Siddiqui and Khurshid, 2012; Qiu et al., 2018) that aim to speed up the overall state space exploration, our technique addresses the problem of checking properties. For each worker, paths that are irrelevant to property checking will be either ignored or quickly processed. Thus, each worker is only responsible to check a small portion of the state space, and the overall time cost of symbolic execution is minimized. Specifically, our work aims at finding assertion violations more efficiently in two aspects: first, one single run of the our technique could find all assertion violations, so users do not have to run potentially long symbolic execution for multiple times; second, if a potential violation is detected, users can get a report about it at the earliest time possible so they can start working on their code at once, even when the checking of other assertions is still going on.

We implemented our technique in an open source framework for symbolic execution, Symbolic Pathfinder (Păsăreanu and Rungta, 2010). To evaluate the effectiveness of our technique, we conducted experiments based on multiple Java artifacts with assertions. The experimental results show that our technique can find all

assertion violations detected by sequential property checking with conventional symbolic execution, while using less time and providing users earlier reports of assertion checking.

Our approach aims at finding assertion violations more efficiently in two aspects: first, one single symbolic execution run could find as many violations as possible, so users do not need to run a potentially long symbolic execution run for multiple times; second, if a potential violation is detected, users can get a report about it at the earliest time possible so they can start working on their code at once, even when the checking of other assertions is still going on.

Our insight of parallel assertion checking among multiple workers is that as a rule, assertions are usually required to be side effect free, which means the checking result of one assertion is not dependent on the checking result of another assertion. Thus, it guarantees that distribution of assertion checking would not change the final result.

Moreover, as a typical embarrassingly parallel problem, the partial result of each worker does not need to be reduced into one overall result. In other words, once the exploration is started, there is very limited amount of data transferring among workers, and complex synchronization mechanism between workers is not necessary. Thus, the partial results could be provided to users as soon as they come out, so users could use them to immediately check the possible problems in the assertion or in the code.

We introduce a novel parallel assertion checking approach with staged symbolic execution, which partitions the exploration into two stages: the first stage explores program code to find out feasible paths to assertions while the second stage checks assertions in parallel guided by the target paths from the first stage. The trie created in the first stage is used to define workloads for parallel workers in the second stage:

- In the first stage, symbolic execution explores program code only to find all feasible paths that lead to the checked assertions. In this stage, we filter out infeasible paths or feasible paths that are irrelevant to assertion checking and save target paths which are used in the second stage with a focus on checking assertions.
- In the second stage, multiple workers are launched in parallel to check assertions guided by the target paths provided by the first stage. Each worker explores the same program guided by a target path using Memoise and checks the assertion that is reached by the target path. All the workers are run in parallel, and individually output the results to users once any of them finishes its job.

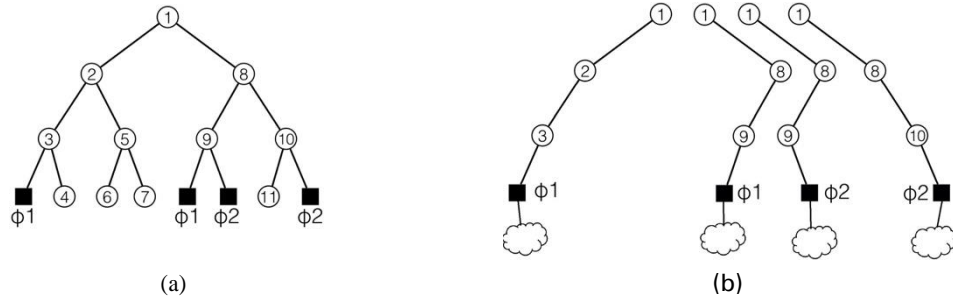


Figure 8: Parallel assertion checking: (a) trie explored by the first stage; (b) paths to be explored by multiple workers for parallel assertion checking.

Figure 8 shows the two-stage analysis on a small subject with two assertions. In the first stage, a symbolic execution run is launched, where a state space trie is built as the exploration is proceeded and four target paths, which are feasible and lead to the checked assertions, are found. We note the state of checking assertion as a frontier state. After gathering all needed information, in the second stage, four workers are launched, each focusing on checking a assertion along one target path, while avoiding the

exploration of irrelevant states and efficiently exploring the four target paths by turning off constraint solving. Once a worker has reached the frontier state, the constraint solver will be turned back on, thus the worker can check the corresponding assertion as well as further exploring the deeper part of the trie from frontier state.

Note that, the two stages do not necessarily have to run sequentially. Instead, as soon as first stage reaches an assertion, we can immediately start the second stage where an available worker is launched to check the reached assertion while the first stage continues looking for other feasible paths to assertions. In other words, the two stages are run simultaneously by multiple workers with different goals.

Ideally, that enough resources are available to allocate for all workers to run at the earliest time possible. However, it is not always true in real execution. Different from the technique we introduced in Chapter III where the number of workers can be calculated by the number of assertions, for this work we do not have a way to predict how many workers we need, for the number of paths towards assertion checking can only be known in run time. Thus, we implement a waiting list mechanism for our solution. The worker of the first stage maintains two lists, one for the path detected and one for idle workers. Once a feasible path towards assertion checking is detected, we first check the worker list and see whether there is an available worker in the second stage. If there is a worker, we assign it with the detected path and start a symbolic execution immediately. If no worker is available at the time, the path information, typically the path choices, will be put into the waiting list. Once a worker in the second stage finished its task, it first checks whether there are any paths in the waiting list. If no path is waiting, worker adds itself to

the idle worker list to be called by the first stage. Otherwise, it will take one of the path information and start a new symbolic execution run with it.

STASE

Algorithms

Algorithm 2 shows the algorithm for finding feasible paths to assertions in the first stage. A regular symbolic execution is applied to explore state space aiming to find assertions to check. This exploration collects the choices processed and maintains the list of choices as the state is advanced or backtracked (Lines 11 and 28). After a state is advanced, the instruction just executed is checked to see it belongs to the list of input assertion instructions. If it is an assertion instruction, the current state is a frontier state that stands for assertion checking. We first check whether there is an idle worker available. If we have such a worker, we launch it for checking this assertion with choice information for the current path (Lines 14-17). Otherwise, we put the detected path into a waiting list (Lines 18-20). Then, the exploration backtracks to find other paths to assertions (Line 21).

Algorithm 2 Algorithm of Finding Feasible Paths to Assertions in the First Stage

Input: Program P , List of assertions $Asserts$, Depth boundary D , List of detected paths $Paths$, List of idol workers $Workers$

Output: Report of symbolic execution R

```
1: Path Choice  $currentPathChoice \leftarrow null$ 
2: Path Choice List  $pathChoices \leftarrow empty$ 
3: Current Depth  $depth \leftarrow 0$ 
4: State  $s \leftarrow \text{root state of } P$ 
5: while true do
6:   if  $d == D$  then
7:     backtrack
8:   end if
9:   if state is advanced then
10:     $s \leftarrow advanced(s)$ 
11:     $pathChoices \leftarrow pathChoices.push(getCurrentChoice())$ 
12:     $depth \leftarrow depth + 1$ 
13:    Instruction  $inst \leftarrow getLastExecutedInstruction()$ 
14:    if  $inst$  is in  $Asserts$  then
15:      if  $Workers$  is not empty then
16:        assign worker  $W$  with  $(P, D, pathChoices)$ 
17:        Remove  $W$  from  $Workers$ 
18:      else
19:        put  $pathChoices$  to  $Paths$ 
20:      end if
21:      backtrack
22:    end if
23:  end if
24:  if state is backtracked then
25:     $s \leftarrow backtracked(s)$ 
26:    if  $s == \text{root state}$  then
27:      break
28:    else
29:       $pathChoices \leftarrow pathChoices.pop()$ 
30:       $depth \leftarrow depth - 1$ 
31:    end if
32:  end if
33: end while
34: return  $R$ 
```

Algorithm 3 shows the algorithm for guided assertion checking in the second stage. Symbolic execution in this stage is guided by the given path choices to quickly traverse through the state space towards the assigned frontier state (Lines 7-10). Constraint solver is turned off since this path has been explored in the first stage and the path conditions along this path are guaranteed to be satisfied until the frontier state is reached. Upon reaching the frontier state, constraint solver is turned on to resume regular symbolic execution to check the corresponding assertion (Line 13). Search is stopped when it backtracks to the frontier state (Line 25-26), and a report is generated regarding the validity of all the checked assertions this worker has encountered (Line 33). Since all workers are run in parallel, earlier feedback could be expected in comparison to sequential property checking using the conventional symbolic execution configured to explore all errors, which only gives an overall report after exploring all the state space. Before the process ends, the worker first checks whether there are paths not assigned in the list that is maintained in the first stage (Line 34). If there is no such a path, the worker adds itself to the idle worker list and prepare to be assigned (Line 35). Otherwise, it gets one of the paths in the waiting least and start a new symbolic execution based on it (Lines 37-38).

Algorithm 3 Algorithm of Guided Assertion Checking in the Second Stage

Input: Program P , List of assertions $Asserts$, Depth Boundary D , Path Choices List $pathChoices$, List of detected paths $Paths$, List of idol workers $Workers$

Output: Report of symbolic execution R

```
1: Current Depth  $depth \leftarrow 0$ 
2: State  $s \leftarrow$  root state of  $P$ 
3: Frontier State  $fs \leftarrow null$ 
4: Choice Generator  $gc \leftarrow null$ 
5: Constraint Solver is turned off
6: while true do
7:   if  $\neg pathChoices.isEmpty()$  then
8:      $guidedExplore(pathChoices.poll())$ 
9:      $s \leftarrow advanced(s)$ 
10:     $depth \leftarrow depth + 1$ 
11:   else
12:      $fs \leftarrow s$ 
13:     Constraint Solver is turned on
14:     while true do
15:       if  $depth == D$  then
16:         backtrack
17:       end if
18:       if state is advanced then
19:          $s \leftarrow advanced(s)$ 
20:          $depth \leftarrow depth + 1$ 
21:       end if
22:       if state is backtracked then
23:          $s \leftarrow backtracked(s)$ 
24:          $depth \leftarrow depth - 1$ 
25:         if  $s == fs$  then
26:           break
27:         end if
28:       end if
29:     end while
30:     break
31:   end if
32: end while
33: return  $R$ 
34: if  $Paths$  is empty then
35:   add current worker to  $Workers$ 
36: else
37:   get path choices  $pathChoices'$  from  $Paths$ 
38:   start current worker with inputs  $(P, D, pathChoices')$ 
39: end if
```

Currently, our approach does not check the reachability of assertions statically. In other words, if there are no assertions at all in the program, the first worker will explore the whole state space in the same way as the conventional symbolic execution, while monitoring the execution of instructions. Even in this case, since the list of assertion instructions is empty, the cost should be similar as conventional symbolic execution, which is acceptable.

Evaluation

Our technique named as STASE, which stands for staged symbolic execution. STASE is intended to find all the assertion violations that could be found by sequential property checking using conventional symbolic execution, while reducing the cost of checking and producing earlier assertion checking reports. We examine the effectiveness and efficiency of STASE relative to sequential property checking using conventional symbolic execution. This leads to the following research questions:

- **RQ1:** How effective is STASE in detecting assertion violations compared to sequential property checking using conventional symbolic execution?
- **RQ2:** How does the cost of applying STASE compared to sequential property checking using conventional symbolic execution on checking assertions?
- **RQ3:** How does the complexity of the checked assertions affect the efficacy of STASE?
- **RQ4:** How does STASE perform when the number of workers is limited?

Tool Support

We implemented our technique in Symbolic Pathfinder (SPF) (Păsăreanu and Rungta, 2010). We have customized Memoise so that the trie nodes corresponding to

assertions are treated as frontier nodes, and each worker leverages Memoise analysis to check the assertion along the target path(s) which are feasible and lead to the checked assertions. Symbolic execution is guided by the target path and constraint solving is turned on only after the frontier node is reached.

We used LoneStar 5 on Texas Advanced Computing Center (TACC) cluster for parallel execution. The configuration of our computing node is:

- Dual Socket
- Xeon E5-2690 v3 (Haswell): 12 cores per socket (24 cores/node), 2.6 GHz
- 64 GB DDR4-2133 (8 x 8GB dual rank x8 DIMMS)
- No local disk
- Hyperthreading Enabled - 48 threads (logical CPUs) per node
- JDK-1.8

We chose the latest version of SPF to support the JDK-1.8. The communicating and allocating of jobs are controlled by TACC operation system and we confirmed its effectiveness.

For the RQ4, we would like to evaluate the performance of STASE when the number of workers is limited. We first finish other evaluations and see the number of workers needed in total for each subject, and then set different number of workers to check the efficiency.

Artifacts

In our evaluation, we used one subject with manually created assertions and three other subjects with assertions synthesized from invariants that are inferred by (Ernst et al., 2007). These subjects have been used in research projects for evaluating the

performance of symbolic execution techniques (Yang et al., 2014b, a, 2012; Staats and Păsăreanu, 2010; Qiu et al., 2018).

The first subject is WBS, which comes with official jpf-symbc package, and has been used in the literature for evaluating symbolic execution techniques. It is a synchronous reactive component from the automotive domain. It consists of one class and 231 lines of code. Since the original WBS code does not have assertions, we used Daikon (Ernst et al., 2007) to dynamically infer invariants for the program, and transformed them into Java assert statements. Since the eight invariants generated by Daikon represent post-conditions, they are all for the exit points of a method. We moved these assertions to different locations of the code which are randomly selected, to simulate the assertions written by users in practice. For this subject, it is not important whether the synthesized assertions are valid or not, as long as the subject can be successfully compiled.

The second subject is an open-source Java subject named trityp. This method takes three integer inputs which stand for the length of three edges of a triangle, and return an integer indicating the type of the triangle (scalene, isosceles, equilateral, or not a triangle). For this subject, ten assertions are manually inserted by others (Yang et al., 2014a).

The third subject is MerArbiter, a component of the flight software for NASA JPL’s Mars Exploration Rovers (MER). This subject has 4.7 KLOC in 268 classes. Similar to trityp, we put 6 assertions generated from Daikon invariants at random locations while ensuring all these assertions are still valid. For this experiment, we would

like to see the performance with a relatively large subject where the assertions are all valid.

The last subject is Apollo (RJC), which is also a Java subject used for evaluation in multiple works (Yang et al., 2014b). It is a Simulink model that was automatically translated to Java with 2.6 KLOC in 54 classes. For this subject, solving constraints is more expensive as they involve nonlinear calculation. We randomly pick up 7 assertions generated by Daikon, which are valid, invalid or irrelevant to the subject, and move them to different locations of the code which were also picked at random.

To evaluate RQ3, we manually increased the complexity of assertions by putting together the conditions from multiple assertions. In this way, the conditions in the newly constructed assertions are more complex than previous assertions. As always, we inserted these assertions at randomly selected locations of the programs while making sure that programs can still compile.

Variables and Measures

The independent variable in our study is the property checking algorithm used in our empirical study. We use the parallel property checking algorithm, and as a control, we use sequential property checking using conventional symbolic execution as implemented in the SPF framework.

For our study, we selected four dependent variables and measures:

1. Time, which is measured as the total elapsed time reported by SPF.
2. States explored, which provides a count of the number of symbolic states generated during symbolic execution.
3. Constraint solver calls, which provides a count of calls made by symbolic

execution to the underlying constraint solver.

4. Property violations, which provides a count of total violations of the checked properties detected during symbolic execution.

For STASE, we collect the four types of information for each worker as well as for the overall process. The total time calculates the elapsed time from the very beginning to the very end of the whole parallel property checking approach. The total number of constraint solver calls is the sum of the number of constraint solver calls made by each worker; likewise, is the total property violations. In contrary, the total states number is not the sum of the number of states explored by each worker; instead, it is the total number of unique states explored by all workers, since the states explored by different workers could overlap.

Results and Analysis

In this section, we present the results of our case study, and analyze the results with respect to our three research questions.

In Table 3, we list the results of running sequential property checking using one run of symbolic execution and STASE on each subject program. For each subject, we list cost including time (in the unit of milliseconds), number of states explored, and number of constraint solver calls made, for performing sequential property checking, and the number of property violations detected by sequential property checking. We also list the cost for STASE, and the number of property violations detected by parallel property checking. For STASE, we list both the cost and violations from each individual worker as well as the total cost and total violations from the whole parallel checking process.

Table 3: Results of checking properties

Subject	Sequential Checking				Parallel Checking				
	Time (ms)	States	Solver Calls	Violations	Workers	Time (ms)	States	Solver Calls	Violations
WBS	612	127	126	12	8	477 (86-303)	127 (4-53)	126 (2-52)	12 (0-4)
trityp	158,033	1,085	1,084	0	97	134,852 (3,182-128,214)	1,085 (13-786)	1,084 (2-188)	0
MerArbiter	171,991	18,783	18,782	0	122	124,497 (514-114,736)	18,783 (7-1,316)	18,782 (3-113)	0
Apollo	82,825	867	866	21	88	27,859 (86-27,412)	867 (54-332)	866 (16-94)	21 (0-8)

- *RQ1: How effective is STASE in detecting assertion violations compared to sequential property checking using conventional symbolic execution?*

As shown in the table, STASE finds the same number of property violations as sequential property checking, despite of the difference in the number of violations across workers. For example, across the 88 workers for checking properties in Apollo, some worker detected 0 violation, while some other worker detected 8 violations, which is more than one third of the total violations detected. For subject MerArbiter, both sequential checking and parallel checking detected 0 property violations.

To further examine this, we report the detailed results for WBS as shown in Table 4. Worker 0 is the one aiming to find feasible paths to all assertions in the first stage, and thus it did not detect any property violations. Workers 1 - 8 are the newly launched workers specifically for checking properties. However, one worker (4) did not find any property violation, five workers (1, 2, 5, 6, 8) each found one property violation, while two workers (3 and 7) found more than 3 violations. In total, all the workers detected 12 violations, that are exactly what sequential property checking found.

Table 4: Results of checking properties in WBS

	Time (ms)	States Explored	Solver Calls	Violations Detected
Sequential Checking	612	127	126	12
Worker 0	303	53	52	0
Worker 1	86	4	2	1
Worker 2	198	16	8	1
Worker 3	217	21	14	3
Worker 4	143	23	10	0
Worker 5	231	33	22	1
Worker 6	261	46	4	1
Worker 7	172	27	12	4
Worker 8	243	15	2	1
Overall	477	127/238	126	12

- *RQ2: How does the cost of applying STASE compared to sequential property checking using conventional symbolic execution on checking assertions compare?*

From Table 3, we find in total parallel checking explored number of states explored and made the same number of constraint solver calls as sequential checking. For example, both checking techniques explored 127 states and made 126 solver calls for subject WBS. However, we find that STASE reduced the time cost due to parallelism. The speedup achieved by parallel checking ranges from 1.17X (for tyityp) to 2.97X (for Apollo). Note that although the cost varies a lot on different workers, the biggest cost of one worker could become the bottleneck. Take trityp as an example, one worker took 128, 214 milliseconds, which is 95% of the total cost.

Again, in Table 4 which shows the details for WBS, we report both the total number of unique states (127) and the total number of states explored (238). Although

there is such difference in states, it is not the case for solver calls. The saving in the number of solver calls is due to the use of memoized symbolic execution, which turns off constraint solving for the path that has already explored before.

Overall, we find that STASE reduced the time cost, while explored the same number of unique states and made the same number of solver calls compared to sequential checking. Moreover, due to the cost reduction as well as parallelism, the users get the property checking results much earlier in parallel checking than in sequential checking.

- *RQ3: How does the complexity of the checked assertions affect the efficacy of STASE?*

Table 5 shows the results of checking the same four programs with more complex assertions using sequential checking versus parallel checking. As we can find in the table, the previous observations for RQ1 and RQ2 remain the same here. In particular, parallel checking detected the same number of property violations as sequential checking. Parallel checking explored the same number of unique states and made the same number of solver calls as sequential checking. Parallel checking also reduced the time cost, but for most cases, the speedup achieved here is more significant than that in the previous study. For example, the speedup achieved for Apollo here is 5.65X, compared to 2.97X which was achieved for the same subject but with simpler assertions. Likewise, it is 3.58X vs. 1.38X for subject MerArbiter. However, we also notice that it is 1.10X vs. 1.17X for subject trityp, where the speedup is getting worse. We conjecture this is due to the fact that one worker took long time (140923 vs. total time 156392) to check the properties and became

the bottleneck. Overall, we see that for 3 out of 4 subjects, parallel checking for the same program but with more complex assertions achieved more significant speedup.

Table 5: Results of checking more complicate properties

Subject	Sequential Checking				Parallel Checking				
	Time (ms)	States	Solver Calls	Violations	Workers	Time (ms)	States	Solver Calls	Violations
WBS	14,018	592	591	42	67	6,712 (95-3,258)	592 (7-267)	591 (2-214)	42 (0-7)
trityp	172,841	1,425	1,424	0	97	156,392 (3,921-140,923)	1,425 (13-1,224)	1,424 (2-193)	0
MerArbiter	918,461	161,231	161,230	0	93	256,216 (612-159,306)	161,231 (6,123-15,123)	161,230 (72-9,122)	0
Apollo	340,427	2,802	2,801	95	88	60,252 (104-59,306)	2,802 (83-725)	2,801 (31-254)	95 (0-27)

- *RQ4: How does STASE perform when the number of workers is limited?*

Finally, we evaluate the performance of STASE when the number of workers we have are limited. Since the results above have already shown that STASE will not miss the number of property violations detected, in this experimental run we only concentrate on the time cost.

Based on the result collected from Table 3 and Table 5, the number of workers needed for each subject (both with simple assertions and complicated assertions) ranges from 8 to 122. Based on these numbers, we conducted three extra groups of experiments with 16, 32 and 64 workers in the second stage. We consider these numbers are mediocre to describe common settings to run parallel programs on a cluster while not too large to run all workers at the same time.

The result of our experiments is shown in Table 6. This table has two parts, where the upper four lines show the result of the subjects with simple assertions and the bottom four lines show the result of subjects with more complex assertions. For each subject, we show the number of workers needed and the time cost of sequential symbolic execution as well as staged symbolic execution with unlimited, 16, 32 or 64 workers.

As we can see from the results, as the number of workers decreases, for most cases the speedup drops slightly with STASE. However, in some of the cases, having a smaller number of workers do not result in a lower speedup. For instance, subject trityp with complex assertions performs as good with 32 workers and 64 workers as with unlimited resources, although the total number of required workers exceeds the configuration. We find the reason being that the paths toward assertion checking in this subject is, even with complicated assertions, relatively simple, so the workers will always finish the task within a reasonably short period of time and thus the idle worker list is never empty. Meanwhile, we have not found any case where the total time increases dramatically with smaller number of workers. This shows that the overhead of communication between workers is small and can be ignored. We consider it an expected result, for we only need to transfer the choices of a path between workers, which can be represented using a stream of small data types. No large block of data needs to be transferred between workers, as we do not need to collect the final result to one certain worker in our technique.

Table 6: Results of checking properties with limited workers

Subject	Sequential	Workload	Unlimited Workers		64 Workers		32 Workers		16 Workers	
	Time (ms)		Time (ms)	Speed-up	Time (ms)	Speed-up	Time (ms)	Speedup	Time (ms)	Speed-up
WBS	612	8	477	1.28x	479	1.28x	476	1.29x	477	1.28x
trityp	150,833	97	134,852	1.12x	135,123	1.12x	135,232	1.12x	145,142	1.04x
MerArbiter	171,991	122	124,497	1.38x	140,245	1.23x	141,523	1.22x	163,512	1.05x
Apollo	82,825	88	27,859	2.97x	28,142	2.94x	31,523	2.63x	35,312	2.35x
WBS	14,018	67	6,712	2.09x	6,714	2.09x	6,881	2.04x	7,162	1.96x
trityp	172,841	97	156,392	1.11x	156,916	1.10x	157,313	1.10x	167,806	1.03x
MerArbiter	918,461	93	256,216	3.58x	281,561	3.26x	326,024	2.82x	359,186	2.56x
Apollo	340,427	88	60,252	5.65x	62,681	5.43x	86,037	3.96x	114,621	2.97x

Threats to validity

The primary threat to external validity for this study involves the representativeness of our object programs. The programs are relatively small, and we only assume and simulate the situation when computing resources are enough to apply a full parallelism. Although this assumption allowed us to check the best efficiency of our technique, these threats still need to be addressed by additional studies on different workloads and run configurations.

The primary threat to the internal validity of this experiment is possible faults when we implemented our technique, and in the tools and environment we selected to perform evaluation. This threat is handled by applying functional tests on our tools on test subjects where we can manually check the correctness of the implementation. A second threat involves inconsistent decisions and practices in the implementation of algorithms and in the execution of evaluation runs, which may lead to accidental unfair comparison with conventional symbolic execution. We controlled this threat by having the algorithms implemented and run by a same developer (the first author), in order to ensure consistency in both implementation and run configurations.

V. CONSTRAINT SOLVING WITH DEEP LEARNING

This chapter presents our work to reuse the constraint satisfiability results with deep learning technique to speed up symbolic execution. Different from the two works introduced in Chapter III and Chapter IV, which are aiming to reduce the impact of path explosion problem in a specific usage of symbolic execution, this approach aims to reduce the cost of constraint solving and can be applied to all symbolic execution techniques.

Overview

As the most time-consuming task in symbolic execution, constraint solving is the key supporting technology that affects the effectiveness of symbolic execution. The advances in constraint solving techniques, for example, by leveraging multiple decision procedures in synergy (De Moura and Bjørner, 2008), have enabled symbolic execution to be applicable to larger programs. However, despite these technological advances, symbolic execution still suffers from the high cost of constraint solving. Several techniques have been developed to speed up constraint solving for symbolic execution by reusing previous solving results (Yang et al., 2012; Visser et al., 2012; Jia et al., 2015; Makhdoom et al., 2014; Hossain et al., 2014). Various forms of results caching are utilized, so that the solutions of path conditions encountered in previous analysis can be reused without calling a constraint solver. As a result, the total number of solver calls as well as the corresponding time cost is reduced. For example, Green (Visser et al., 2012) uses an in-memory database Redis (Red, 2019) to store path conditions and their constraint solutions as key-value pairs, in which key is a path condition string and value is a Boolean value showing whether the corresponding path condition is satisfiable or not,

and reuses constraint solutions based on string matching. GreenTrie (Jia et al., 2015) further improves the reuse rate of previous constraint solutions by applying logical reduction and logical subset and superset querying for given constraints. However, such reuse techniques require syntactic/semantic equivalence or implication relationship between constraints. If the equivalence or implication relationship is not satisfied, these techniques are not able to reuse previous constraint solutions.

Deep learning is a popular machine learning technique which has found many applications (Krizhevsky et al., 2017; Socher et al., 2013; Mohamed et al., 2012; Tolstikhin et al., 2018). In deep learning, deep neural networks are trained by using a large set of labeled data, and learn to perform classification tasks directly from text, images or sound. It has been shown to be effective and efficient for difficult classification problems such as image classification (He et al., 2015) and speech recognition (Mohamed et al., 2012). In this paper, we introduce DeepSolver, a novel approach to constraint solving with deep learning for symbolic execution. In our approach, with the help of canonization and vectorization, path conditions are represented with informative and discriminating features for satisfiability classification, which are then extracted by deep learning. Instead of reusing each individual constraint solution, DeepSolver utilizes collective knowledge of constraint solutions to train deep neural networks, and then use the trained deep neural networks to classify path conditions for satisfiability during symbolic execution. We evaluate the accuracy and the efficiency of DeepSolver in classifying path conditions using 12 Java subjects, and compare DeepSolver with Z3, Green, and GreenTrie, the state-of-the-art constraint solving and constraint solution reuse

techniques. We also evaluate how DeepSolver supports symbolic execution compared to GreenTrie.

DeepSolver

In this section, we present DeepSolver, a novel approach to constraint solving with deep learning for symbolic execution. DeepSolver consists of two stages: training DNNs with constraint solutions and classifying path conditions with DNNs for satisfiability.

Training DNNs with Constraint Solutions

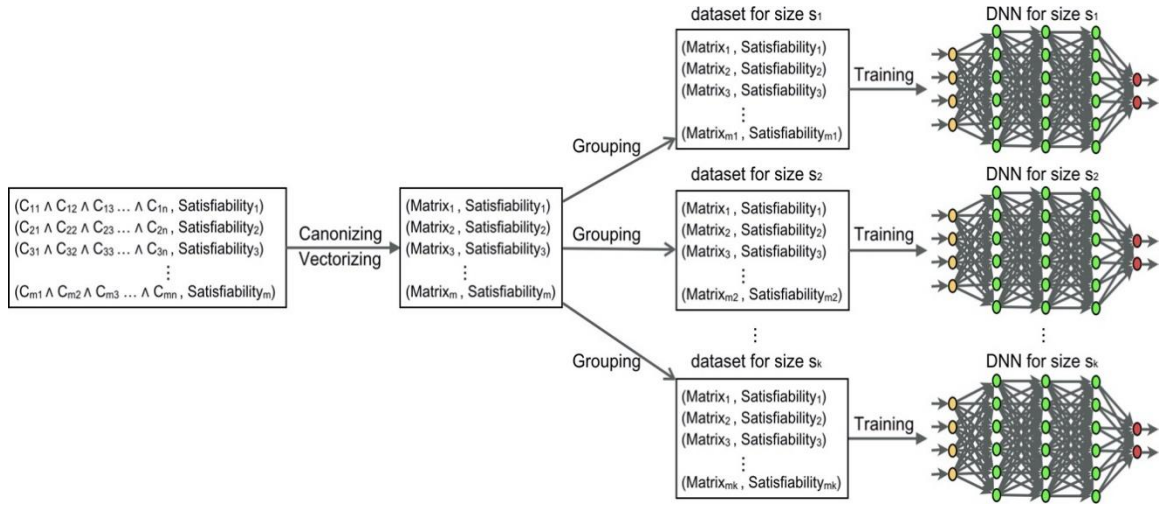


Figure 9: Training DNNs with constraint solutions

Figure 9 shows the overall process of training DNNs with constraint solutions. First, we organize the existing constraint solutions using PC-satisfiability pairs where PC is a path condition and satisfiability is a Boolean value (i.e., True or False) indicating whether the PC is satisfiable or not. Then we canonize and vectorize each path condition into a matrix, as DNNs require input data to be in the form of matrices. As shown in Figure 9, all the PC-satisfiability pairs are transformed into matrix-satisfiability pairs. Next, we group the matrix-satisfiability pairs into multiple datasets based on the size of

each matrix. Matrices with the same number of rows and the same number of columns will be grouped together into a dataset, which will be used to train a specific DNN. And this particular DNN later will be selected to classify path conditions, whose matrix forms possess the same size as the input matrices here in training this DNN.

Canonizing

Path conditions generated during symbolic execution do not have a common form by default. For instance, the name space of symbolic variables differs from subject to subject. *Canonizing* follows the standard approach to transform a path condition into a normal form and helps eliminate equivalent records in training datasets. Specifically, for a linear integer arithmetic path condition (conjunction of constraints), each constraint in the path condition is transformed into a normal form as $c_0v_0 + c_1v_1 + c_2v_2 + \dots + k \text{ op } 0$, where c_n is the coefficient of variable c_n , k is the constant term, and $\text{op} \in \{=, \neq, \leq\}$. We handle constraints with $=$, \neq , or \leq by themselves, but constraints with other operators such as $>$, $<$ and \geq are transformed into the canonical form over \leq . Meanwhile, constraints are sorted in a lexicographic order and symbolic variables are renamed based on their appearances in the path condition from left to right. For instance, both $PC1: x + y < z \wedge x = z \wedge x - 10 > y$ and $PC2: a + b < c \wedge a = c \wedge a - b > 10$ are canonized into the same shape $v_0 + v_1 - v_2 + 1 \leq 0 \wedge v_0 - v_2 = 0 \wedge -v_0 + v_1 + 11 \leq 0$.

Vectorizing

Vectorizing is to transform a path condition into a 2-dimensional matrix. After canonizing, each constraint is in the form of $c_0v_0 + c_1v_1 + c_2v_2 + \dots + k \text{ op } 0$.

Therefore, a path condition can be vectorized into a matrix as follows: each row in the

matrix represents a constraint, and each column in the matrix stands for the coefficient of each symbolic variable, the constant term, or the integer value representing op. Due to this simple representation of constraints as respective rows of a matrix, DeepSolver supports any number of constraints, say n - as long as there is a training dataset of path conditions with n constraints.

Algorithm 4 shows how to vectorize a canonized path condition into a matrix. We first initialize *Matrix* by the number of constraints and the largest index of symbolic variable in the path condition (Lines 1-3). The path condition is split by “ \wedge ” into a list of constraints *BCS* (Line 4). Each constraint in *BCS* is checked to setup a row in *Matrix* (Lines 5-24). For each constraint, we first check its operator and set the corresponding item in the row as 0, 1 or 2 (Lines 7-13). we assign value 0 for $=$, 1 for \neq , and 2 for \leq , respectively. Then the constraint is further broken down to a list of terms *Terms* after removing the equation operator op (Lines 14-15) As we go through each term in *Terms*, if the term is in a shape of $c_j \times v_j$, we set the j -th item in the row as c_j (Lines 17 - 18); otherwise, the term is a constant value k , which is set as the value of the second last item in the row (Lines 20-21).

After all constraints are processed, *Matrix* is the final vectorized result of the path condition (Line 25). For instance, the previous example path condition $v0 + v1 - v2 + 1 \leq 0 \wedge v0 - v2 = 0 \wedge -v0 + v1 + 11 \leq 0$ can be transformed into a matrix with size 3×5 as follows:

$$\begin{bmatrix} 1 & 1 & -1 & 1 & 2 \\ 1 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 11 & 2 \end{bmatrix}$$

Algorithm 4 Algorithm of Vectorizing a Canonized PC into a Matrix

Input: A canonized path condition PC , which is linear and in shape of $BC_0 \wedge BC_1 \wedge \dots \wedge BC_m$, where BC_m is in shape of $c_0v_0 + c_1v_1 + c_2v_2 + \dots + k \text{ op } 0$ ($op \in \{=, \neq, \leq\}$)

Output: Vectorized PC as matrix $Matrix$

```
1:  $X \leftarrow m + 1$ 
2:  $Y \leftarrow n + 3$ 
3:  $\text{Array}[X][Y] \text{ Matrix} \leftarrow \text{empty}$ 
4:  $\text{List } BCS \leftarrow PC \text{ split by " } \wedge "$ 
5:  $i \leftarrow 0$ 
6: while  $i < X$  do
7:   if  $op$  in  $BCS[i]$  is " $=$ " then
8:      $Matrix[i][Y - 1] \leftarrow 0$ 
9:   else if  $op$  in  $BCS[i]$  is " $\neq$ " then
10:     $Matrix[i][Y - 1] \leftarrow 1$ 
11:  else if  $op$  in  $BCS[i]$  is " $\leq$ " then
12:     $Matrix[i][Y - 1] \leftarrow 2$ 
13:  end if
14:   $BCS[i] \leftarrow BCS[i] \text{ remove } op$ 
15:   $\text{List } Terms \leftarrow BCS[i] \text{ split by " } + "$ 
16:  for all  $term$  in  $Terms$  do
17:    if  $term$  in shape of  $c_j \times v_j$  then
18:       $Matrix[i][j] \leftarrow c_j$ 
19:    else
20:       $\{term \text{ is the constant term } k\}$ 
21:       $Matrix[i][Y - 2] \leftarrow k$ 
22:    end if
23:  end for
24: end while
25: return  $Matrix$ 
```

Grouping

Now all the PC-satisfiability pairs are transformed into *matrix-satisfiability* pairs.

As shown in Figure 9, these *matrix-satisfiability* pairs are then regrouped according to the size of each matrix, i.e., the matrices that have the same number of rows and the same number of columns will form a training dataset as input data to train the corresponding DNN.

Training

Classifying path conditions is a binary classification problem, in which the labeled training data sample belongs to either one of the two known classes or categories, i.e., satisfiable and unsatisfiable. Class imbalance occurs when one class (the majority group) contains significantly more samples than the other class (the minority group), and could cause bias towards the majority class and may ignore the minority class altogether in extreme cases (Johnson and Khoshgoftaar, 2019).

In order to ensure the efficacy of DNNs in classifying path conditions, if a training dataset has the class imbalance problem, it needs to be balanced before being used to train a DNN. How to balance a training dataset is an important area of research. Essentially, there are three categories of methods for handling class imbalance in machine learning: data-level techniques, algorithm-level methods, and hybrid approaches (Krawczyk, 2016). In particular, data-level techniques use data sampling methods to reduce the level of imbalance. In our work, we use two particular methods to add more data to the minority group. One method is using mechanical transformations or mutation analysis (Acree et al., 1979). For example, we can create unsatisfiable path conditions from an existing satisfiable path condition by randomly mutating coefficients of variables to a randomly generated value or mutating the relational operator to one of the other two operators. The mutated path conditions are then sent to a constraint solver to check satisfiability. If unsatisfiable, it is added to the training dataset; otherwise, it is discarded. The other method is using Generative Adversarial Networks (GAN) (Goodfellow et al., 2014) to generate new path conditions. In a GAN, we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that

estimates the probability that a sample came from the training data rather than G. The training stops when G becomes capable to generate data that could confuse D, which means the later one could not tell whether the data are real or generated. In our case, we use the collected path conditions to train a GAN framework and use the generator to create more path conditions that are supposed to be similar to the data collected from real subjects. Similar to the mutation-based method, we send the GAN-generated path conditions to the solver and add to the training dataset only the path conditions that belong to a minority group. As illustrated in Figure 9, multiple DNNs are trained with existing constraint solutions, and will be used to classify path conditions.

Classifying Path Conditions Using DNNs

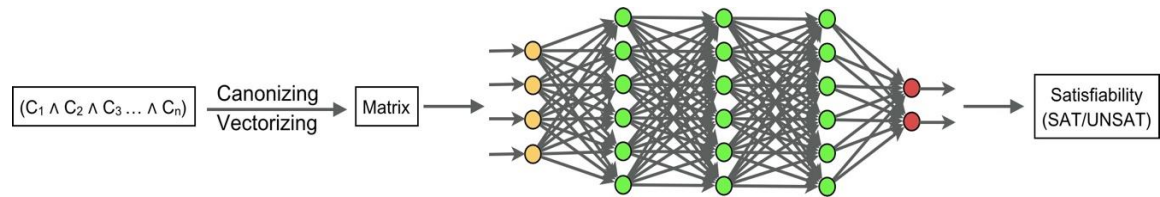


Figure 10: Classifying a path condition using a DNN

Figure 10 shows the steps to classify a path condition generated in symbolic execution with a DNN that has been trained with existing constraint solutions. The path condition is canonized and vectorized as discussed in the training stage and is transformed into a matrix. Based on the size of the matrix, one of a previously trained DNNs is retrieved to classify this path condition. The classification result (satisfiable or unsatisfiable) is then returned to symbolic execution to decide whether the corresponding path is feasible or not. If no DNN is applicable for the given matrix, a regular constraint solver is called to solve the path condition. As long as there is a previously trained DNN that matches the size of the matrix of a path condition, our approach is capable of

classifying the path condition for its satisfiability. Moreover, since DNNs are trained off-line, users can train different DNNs as needed while the classification with existing DNNs is still in progress.

Symbolic Execution with DeepSolver

It is difficult for DNNs to reach 100% accuracy in classification. In fact, 100% accuracy indicates the possibility of over-fitting problem Cawley (2012), which happens when a DNN is overly refined to a certain dataset and thus is unable to classify other inputs while keeping high accuracy. As a result, DNNs are usually used with high accuracy while tolerating potential misclassification. DeepSolver has two types of potential misclassification: satisfiable path conditions are misclassified as unsatisfiable (SAT-To-UNSAT misclassification) or unsatisfiable path conditions are misclassified as satisfiable (UNSAT-To-SAT misclassification). We discuss in the following our strategies for addressing these two types of misclassification to ensure the soundness of symbolic execution with DeepSolver.

SAT-To-UNSAT Misclassification

When a satisfiable path condition is misclassified as unsatisfiable, the corresponding path is incorrectly identified as infeasible, and symbolic execution will not continue the exploration along this path. This misclassification causes symbolic execution to explore fewer states and must be avoided. To address this problem, we propose to double check the classification result when a path condition is classified by DeepSolver as unsatisfiable, by calling a conventional constraint solver. To the best of our knowledge, for most programs, the number of unsatisfiable path conditions encountered by symbolic execution is much smaller than the number of satisfiable path

conditions. Therefore, as long as we ensure DeepSolver works with high classification accuracy, the chance of having a path condition misclassified by DeepSolver as unsatisfiable is small, and the overhead introduced by calling a conventional constraint solver is acceptable.

UNSAT-To-SAT Misclassification

When an unsatisfiable path condition is classified as satisfiable, the corresponding path is incorrectly identified as feasible, and symbolic execution will continue exploring states that are in fact not feasible. For intermediate states, we may ignore this type of misclassification based on the following two observations: first, it is safe to explore infeasible states; second, state exploration along this misclassified feasible path will quickly stop when DeepSolver’s classification accuracy is relatively high. More specifically, assuming DeepSolver’s accuracy in classifying unsatisfiable path conditions is 90%, the chance of UNSAT-To-SAT Misclassification is 10%. When such misclassification happens on an infeasible path with a path condition PC , symbolic execution will continue to explore the next infeasible path with path condition updated to $PC' = PC \wedge c$, where c is the new constraint collected along the path. As PC and PC' are vectorized to two matrices with different sizes, they are classified independently with two DNNs; thus, the chance of misclassifying both of them is $10\% \times 10\% = 1\%$. For the same reason, the chance of three consecutive UNSAT-To-SAT misclassification is $10\% \times 10\% \times 10\% = 0.1\%$. Therefore, the chance of continuous UNSAT-TO-SAT misclassification drops significantly as the exploration goes deeper. In other words, even if an unsatisfiable path condition is misclassified as satisfiable, it is very likely that the exploration will only explore very few infeasible states before DeepSolver will make

correct classification (and symbolic execution will backtrack). For leaf states, which represent complete paths or paths stopped due to errors, the conventional constraint solver is called to find input values for the two most popular applications of symbolic execution: test case generation and error detection.

Algorithm

Algorithm 5 Algorithm of Symbolic Execution with DeepSolver

Input: Previously trained DNN model collection M

Output: Test suit T

```

1: Test Suit  $T \leftarrow \emptyset$ 
2:  $init\_state.PC \leftarrow True$ 
3:  $stack.push(init\_state)$ 
4: Boolean  $\varphi \leftarrow True$ 
5: while  $\neg stack.empty()$  do
6:    $s \leftarrow stack.pop()$ 
7:    $\varphi \leftarrow deepSolve(s.PC, M)$ 
8:   if  $\varphi$  is False or  $s.PC$  is not supported by  $M$  then
9:      $\varphi \leftarrow solve(s.PC)$ 
10:  end if
11:  if  $\varphi$  is True then
12:    while  $inst \neq null$  do
13:      if  $inst$  is if( $c$ ) then
14:        {Let  $c$  be constraint for True branch}
15:         $s'.PC \leftarrow s.PC \wedge c$ 
16:         $stack:push(s')$ 
17:         $s''.PC \leftarrow s.PC \wedge \neg c$ 
18:         $stack:push(s'')$ 
19:        break
20:      else if  $inst$  is abort or halt then
21:        Test case  $t \leftarrow solve(s.PC)$ 
22:         $T \leftarrow T \cup \{t\}$ 
23:        break
24:      else
25:         $s \leftarrow execute(inst, s)$ 
26:         $inst \leftarrow getNextInstruction()$ 
27:      end if
28:    end while
29:  end if
30: end while
31: return  $T$ 

```

Algorithm 5 presents the key steps in symbolic execution with DeepSolver. Similar to traditional forward symbolic execution, it uses depth-first search to explore all feasible program paths. In particular, Lines 7-9 first use DeepSolver to classify the current path condition, and then use the conventional constraint solver to solve the path condition if DeepSolver classifies it as unsatisfiable (double check to avoid misclassification) or if DeepSolver does not support the path condition (i.e., matrix size does not match). On the other hand, if DeepSolver classifies the path condition as satisfiable, the execution continues until a conditional instruction is encountered or the path is naturally completed or aborted due to errors (Lines 12-28). For a conditional instruction, the execution forks for each branch, and the path condition is updated with different conditions accordingly (Lines 13-19). When the path is completed or aborted, the conventional constraint solver is called to generate test values for the completed or aborted paths (Lines 20-23).

Evaluation

This section evaluates DeepSolver on its performance in classifying path conditions and in supporting symbolic execution. The evaluation aims to answer the following five research questions:

- **RQ1:** How accurate is DeepSolver in classifying path conditions?
- **RQ2:** How do mutation and GAN based balancing methods impact the accuracy of DeepSolver?
- **RQ3:** How efficient is DeepSolver in classifying path conditions?
- **RQ4:** How does the number of hidden layers/neurons impact DeepSolver's efficacy in classifying path conditions?

- **RQ5:** How efficient is symbolic execution with DeepSolver?

We use baseline techniques including Z3 (Z3S, 2019) for conventional constraint solving, Green (Visser et al., 2012) and GreenTrie (Jia et al., 2015) for constraint solution reuse.

Implementation and Subjects

We implemented canonizing and vectorizing modules, and symbolic execution with DeepSolver in Symbolic Pathfinder (SPF) (Păsăreanu and Rungta, 2010), a widely used open-source symbolic execution framework for Java programs. We trained DNNs with Keras (Chollet et al., 2015), which is a high-level deep learning API written in Python and ran it on top of TensorFlow (Abadi et al., 2015).

The subjects chosen for our evaluation have been widely used as benchmarks for evaluating symbolic execution techniques (Albert et al., 2011; Inkumsah and Xie, 2008; Yang et al., 2014b, 2012; Souza et al., 2011; Rojas and Păsăreanu, 2013; Yang et al., 2014a; Qiu et al., 2015; Burnim et al., 2009; Jia et al., 2015):

- **Traffic Anti-Collision Avoidance System (TCAS)** is a Java version of a classic anti-collision system available from the SIR repository (SIR). Its code in C together with 41 mutants are available at SIR repository (SIR). We manually converted the code to Java and only used the original version for this case study.
- **Wheel Brake System (WBS)** is a Java version of the synchronous reactive component from the automotive domain, which is used to determine how much braking pressure to apply based on the environment. The Java model is based on a Simulink model derived from the WBS case example found in ARP 4761 (SAE-ARP4761, 1996; Joshi and Heimdahl, 2005). The Simulink model was translated

to C using tools developed at Rockwell Collins and manually translated to Java.

- **MerArbiter** is a component of the flight software for NASA JPL's Mars Exploration Rovers (MER). This subject has 4.7K LOC in 268 classes.
- **Red-Black Tree** is an implementation of the red-black tree data structure from JDK 1.5. Red-Black Tree contains 7 symbolic elements.
- **MergeSort**, **QuickSort**, and **HeapInsert** are three commonly used sorting algorithms from JDK 1.5.
- **Dijkstra** is a benchmark program for finding the shortest paths between nodes in a graph, which may represent road networks for instance, developed by Jacob Burnim from University of California, Berkeley. It contains 109 LOC in one class.
- **TSP** is a benchmark program for traveling salesman problem, developed by Sudeep Juvekar and Jacob Burnim from California, Berkeley. It contains 124 LOC in one class.
- **Rational** is a case study for computing greatest common divisor and its related operations on rational numbers. It contains 96 LOC in one class.
- **BinTree** implements a binary search tree with element insertion, deletion. This subject is used to evaluate GreenTrie and distributed as a benchmark.
- **BinomialHeap** is a Java implementation of binomial heap.

All relevant research artifacts, including subjects, training data, and DNNs generated in the experiments, etc. are publicly available for download (Dee, 2020).

DNN Training

We ran SPF with Z3 on TCAS, WBS and MerArbiter, and collected all the encountered constraint solutions for training DNNs. In total, we collected 514, 230 constraint solutions, which were then canonized, vectorized, and grouped into different datasets. Matrices with the same size, decided by the number of columns and the number of rows, are put in the same dataset.

A training dataset for deep learning usually has thousands of records. For example, UCF-101 (Soomro et al., 2012) has 13K videos, and HMDB-51 (Kuehne et al., 2011) has 6.8K videos. Therefore, to maintain the efficacy of DNNs in DeepSolver, we fixed the size of column to 22 (20 columns for coefficients of 20 symbolic variables, 1 column for the constant term, and 1 column for the relational operator), such that most datasets after grouping have reasonable number of records for training purpose. Matrices with less than 22 columns are enlarged by adding columns with all 0s. For instance, the path condition $v0 + v1 - v2 + 1 \leq 0 \wedge v0 - v2 = 0 \wedge -v0 + v1 + 11 \leq 0$ can be transformed into a matrix with size 3×5 :

$$\begin{bmatrix} 1 & 1 & -1 & 1 & 2 \\ 1 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 11 & 2 \end{bmatrix}$$

This matrix can also be enlarged to a matrix with size 3×6 as follows.

$$\begin{bmatrix} 1 & 1 & -1 & 0 & 1 & 2 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 11 & 2 \end{bmatrix}$$

Since the 4th column represents the coefficients of the 4th symbolic variable in

three constraints, and all 0s indicate that variable is not existent in the path condition. Similarly, during classification stage, if the matrix of a path condition has less than 22 columns, it will get enlarged to 22 columns before classification.

While DeepSolver supports arbitrary number of constraints, the size of matrix used in our evaluation ranges from 11×22 to 28×22 for two reasons: first, our training data have path conditions with 28 constraints at most; second, the training datasets of path conditions with less than 11 constraints have less than 2000 records, which are not enough to train effective DNNs, and thus we decided not to train DNNs for these datasets. In the following, we use the size of a path condition to refer to the size of its corresponding matrix.

Class imbalance occurred in our datasets. In particular, unsatisfiable path conditions were much fewer than satisfiable path conditions in each dataset. This is reasonable since most programs have more feasible paths than infeasible paths explored by symbolic execution, due to the fact that symbolic execution continues exploring along feasible paths but backtracks when the path becomes infeasible. Therefore, we used the methods mentioned earlier in Section V to balance the datasets. We first apply mutation to balance our datasets, in which new path conditions are generated and solved in parallel to speed up this process until we have 45% - 55% unsatisfiable path conditions in the training dataset. In order to answer RQ2, we also train a group of GAN frameworks based on the original datasets and apply the generators to balance the datasets. We stop the generation when the numbers of unsatisfiable path conditions are the same as in the datasets balanced by mutation. Table 7 shows the records in each training dataset before and after balancing.

Table 7: Number of records before and after balancing

Size	Before Balancing			After Balancing		
	SAT	UNSAT	Total	SAT	UNSAT	Total
11x22	3,894	2	3,896	3,894	4,283	8,177
12x22	7,503	4	7,507	7,503	7,503	15,006
13x22	10,590	6	10,596	10,590	10,590	21,180
14x22	14,673	11	14,684	14,673	16,140	30,813
15x22	19,019	24	19,043	19,019	17,117	36,136
16x22	25,526	68	25,594	25,526	22,973	48,499
17x22	33,975	144	34,119	33,975	37,372	71,347
18x22	42,545	537	43,082	42,545	38,290	80,835
19x22	49,917	1,356	51,273	49,917	54,908	104,825
20x22	56,262	800	57,062	56,262	56,262	112,524
21x22	56,158	1,084	57,242	56,158	61,773	117,931
22x22	52,512	780	53,292	52,512	47,260	99,772
23x22	46,042	354	46,396	46,042	41,437	87,479
24x22	36,681	195	36,876	36,681	36,681	73,362
25x22	24,705	63	24,768	24,705	27,175	51,880
26x22	15,934	66	16,000	15,934	14,340	30,274
27x22	9,201	15	9,216	9,201	10,121	19,322
28x22	3,581	3	3,584	3,581	3,581	7,162

To evaluate RQ4, we used two DNN structures, small structure and large structure, according to the number of hidden layers and the number of neurons in each layer. The small structure (5×5) has 5 hidden layers and 5 neurons in each layer, and the large structure (10×10) has 10 hidden layers and 10 neurons in each layer. Both structures use dense connection with ReLU (Agarap, 2018), which is a widely used activation function in deep learning.

Results and Analysis

- *RQ1: How accurate is DeepSolver in classifying path conditions?*

We ran SPF with Z3 on the nine remaining subjects that were not used for training DNNs, collected path conditions solved by Z3, and then applied DeepSolver to classify these path conditions. Since our previously trained DNNs only support path conditions sized from 11×22 to 28×22 , SPF was run with a depth bound 28 and we collected only path conditions with 11-28 constraints for this experiment. Path conditions were grouped according to their vectorized matrix size and were classified by the corresponding DNN. We used Z3’s solving results as ground truth to compute the accuracy of each DNN.

Table 8 shows the results using DNNs trained by datasets that balanced via mutation. It gives the number of satisfiable path conditions (# PC-SAT), unsatisfiable path conditions (# PC-UNSAT), and total path conditions (# PC-Total). For each group of path conditions, it gives the number of path conditions that were correctly classified (# Correct Classification), and the corresponding accuracy (i.e., $\frac{\# \text{ Correct Classification}}{\# \text{ PC}}$).

Similarly, we also report the data when using DNNs balanced via GAN in Table 9. The results show that the accuracy of DeepSolver is consistently high across DNNs for different sizes of path conditions. In Table 8, the accuracy is always over 95% in classifying satisfiable path conditions, while it is always over 83% in classifying unsatisfiable path conditions. For example, the accuracy is 98.9% in classifying satisfiable path conditions for size 16×22 . In Table 9, we observe the accuracy being similar with the data in Table 9, where the accuracy is always over 93% in classifying satisfiable path conditions and is always over 82% in classifying unsatisfiable path

conditions. We observe that the accuracy in classifying unsatisfiable path conditions is slightly worse than classifying satisfiable path conditions. We speculate that this accuracy difference is caused by the difference in training data: while satisfiable constraint solutions were all collected from symbolic execution on real subjects, a portion of unsatisfiable constraint solutions were generated through mechanical transformation on existing constraint solutions, which may not be as diverse as satisfiable constraint solutions. This problem may be addressed in future by collecting more unsatisfiable constraint solutions from real subjects. Nevertheless, we note the overall accuracy of DeepSolver is still high.

Table 8: Accuracy of DeepSolver (Balanced via Mutation) in classifying path conditions

Size	# PC			5x5 DNN						10x10 DNN					
				# Correct Classification			Accuracy			# Correct Classification			Accuracy		
	SAT	UNSAT	Total	SAT	UNSAT	Total	SAT	UNSAT	Overall	SAT	UNSAT	Total	SAT	UNSAT	Overall
11x22	674	37	711	658	31	689	97.6%	83.8%	96.9%	653	33	686	96.9%	89.2%	96.5%
12x22	982	32	1,014	937	28	965	95.4%	87.5%	95.2%	959	29	988	97.7%	90.6%	97.4%
13x22	1,435	29	1,464	1,392	27	1,419	97.0%	93.1%	96.9%	1,377	25	1,402	96.0%	86.2%	95.8%
14x22	2,215	71	2,286	2,178	63	2,241	98.3%	88.7%	98.0%	2,181	65	2,246	98.5%	91.5%	98.3%
15x22	3,536	272	3,808	3,441	232	3,673	97.3%	85.3%	96.5%	3,408	242	3,650	96.4%	89.0%	95.9%
16x22	5,492	119	5,611	5,432	104	5,536	98.9%	87.4%	98.7%	5,305	102	5,407	96.6%	85.7%	96.4%
17x22	7,110	165	7,275	6,961	141	7,102	97.9%	85.5%	97.6%	6,989	149	7,138	98.3%	90.3%	98.1%
18x22	8,313	177	8,490	7,973	156	8,129	95.9%	88.1%	95.7%	8,046	162	8,208	96.8%	91.5%	96.7%
19x22	10,392	416	10,808	9,914	378	10,292	95.4%	90.9%	95.2%	10,225	356	10,581	98.4%	85.6%	97.9%
20x22	8,189	630	8,819	7,796	567	8,363	95.2%	90.0%	94.8%	7,886	564	8,450	96.3%	89.5%	95.8%
21x22	8,311	198	8,509	8,187	172	8,359	98.5%	86.9%	98.2%	7,945	176	8,121	95.6%	88.9%	95.4%
22x22	7,555	328	7,883	7,253	286	7,539	96.0%	87.2%	95.6%	7,252	288	7,540	96.0%	87.8%	95.6%
23x22	6,487	135	6,622	6,241	115	6,356	96.2%	85.2%	96.0%	6,370	120	6,490	98.2%	88.9%	98.0%
24x22	5,221	163	5,384	4,960	140	5,100	95.0%	85.9%	94.7%	5,053	139	5,192	96.8%	85.3%	96.4%
25x22	3,156	73	3,229	3,112	67	3,179	98.6%	91.8%	98.5%	3,058	65	3,123	96.9%	89.0%	96.7%
26x22	2,083	43	2,126	2,023	38	2,061	97.1%	88.4%	96.9%	2,003	38	2,041	96.2%	88.4%	96.0%
27x22	1,330	102	1,432	1,291	94	1,385	97.1%	92.2%	96.7%	1,303	92	1,395	98.0%	90.2%	97.4%
28x22	474	28	502	456	26	482	96.2%	92.9%	96.0%	455	24	479	96.0%	85.7%	95.4%

Table 9: Accuracy of DeepSolver (Balanced via GAN) in classifying path conditions

Size	# PC			5x5 DNN						10x10 DNN					
				# Correct Classification			Accuracy			# Correct Classification			Accuracy		
	SAT	UNSAT	Total	SAT	UNSAT	Total	SAT	UNSAT	Overall	SAT	UNSAT	Total	SAT	UNSAT	Overall
11x22	674	37	711	639	34	673	94.8%	91.9%	94.7%	637	33	670	94.5%	89.2%	94.2%
12x22	982	32	1,014	937	29	966	95.4%	90.6%	95.3%	926	27	953	94.3%	84.4%	94.0%
13x22	1,435	29	1,464	1,374	24	1,398	95.7%	82.8%	95.5%	1,350	26	1,376	94.1%	89.7%	94.0%
14x22	2,215	71	2,286	2,141	63	2,204	96.7%	88.7%	96.4%	2,164	62	2,226	97.7%	87.3%	97.4%
15x22	3,536	272	3,808	3,431	252	3,683	97.0%	92.6%	96.7%	3,419	232	3,651	96.7%	85.3%	95.9%
16x22	5,492	119	5,611	5,286	105	5,391	96.2%	88.2%	96.1%	5,315	110	5,425	96.8%	92.4%	96.7%
17x22	7,110	165	7,275	6,996	147	7,143	98.4%	89.1%	98.2%	6,955	152	7,107	97.8%	92.1%	97.7%
18x22	8,313	177	8,490	7,854	149	8,003	94.5%	84.2%	94.3%	7,934	162	8,096	95.4%	91.5%	95.4%
19x22	10,392	416	10,808	9,677	351	10,028	93.1%	84.4%	92.8%	9,939	352	10,291	95.6%	84.6%	95.2%
20x22	8,189	630	8,819	8,015	528	8,543	97.9%	83.8%	96.9%	7,808	570	8,378	95.3%	90.5%	95.0%
21x22	8,311	198	8,509	8,065	179	8,244	97.0%	90.4%	96.9%	8,184	177	8,361	98.5%	89.4%	98.3%
22x22	7,555	328	7,883	7,210	274	7,484	95.4%	83.5%	94.9%	7,369	285	7,654	97.5%	86.9%	97.1%
23x22	6,487	135	6,622	6,391	122	6,513	98.5%	90.4%	98.4%	6,403	119	6,522	98.7%	88.1%	98.5%
24x22	5,221	163	5,384	5,094	143	5,237	97.6%	87.7%	97.3%	5,013	147	5,160	96.0%	90.2%	95.8%
25x22	3,156	73	3,229	3,088	65	3,153	97.8%	89.0%	97.6%	3,040	65	3,105	96.3%	89.0%	96.2%
26x22	2,083	43	2,126	1,977	36	2,013	94.9%	83.7%	94.7%	2,001	39	2,040	96.1%	90.7%	96.0%
27x22	1,330	102	1,432	1,297	89	1,386	97.5%	87.3%	96.8%	1,305	94	1,399	98.1%	92.2%	97.7%
28x22	474	28	502	468	25	493	98.7%	89.3%	98.2%	452	25	477	95.4%	89.3%	95.0%

- *RQ2: How do mutation and GAN based balancing methods impact the accuracy of DeepSolver?*

To answer RQ2, we evaluate all DNNs trained with both mutation-based balanced datasets and GAN-based balanced datasets. We report both the accuracy in Table 10, and the data loss in Table 11. From these two tables, we observe that the two dataset balancing methods the original training datasets have little impact on the trained DNNs in their performance in classifying path conditions. In general, all DNNs reached high accuracy and small data loss in classifying path conditions, while the accuracy of classifying UNSAT path conditions dropped as we discussed above. In other words, the two methods can be considered equally effective in balancing the training datasets.

Table 10: Comparison of Accuracy between Mutation and GAN Balanced Datasets

Size	Mutation-balanced Training Datasets						GAN-balanced Training Datasets					
	5x5 DNN			10x10 DNN			5x5 DNN			10x10 DNN		
	SAT	UNSAT	Overall	SAT	UNSAT	Overall	SAT	UNSAT	Overall	SAT	UNSAT	Overall
11x22	97.6%	83.8%	96.9%	96.9%	89.2%	96.5%	94.8%	91.9%	94.7%	94.5%	89.2%	94.2%
12x22	95.4%	87.5%	95.2%	97.7%	90.6%	97.4%	95.4%	90.6%	95.3%	94.3%	84.4%	94.0%
13x22	97.0%	93.1%	96.9%	96.0%	86.2%	95.8%	95.7%	82.8%	95.5%	94.1%	89.7%	94.0%
14x22	98.3%	88.7%	98.0%	98.5%	91.5%	98.3%	96.7%	88.7%	96.4%	97.7%	87.3%	97.4%
15x22	97.3%	85.3%	96.5%	96.4%	89.0%	95.9%	97.0%	92.6%	96.7%	96.7%	85.3%	95.9%
16x22	98.9%	87.4%	98.7%	96.6%	85.7%	96.4%	96.2%	88.2%	96.1%	96.8%	92.4%	96.7%
17x22	97.9%	85.5%	97.6%	98.3%	90.3%	98.1%	98.4%	89.1%	98.2%	97.8%	92.1%	97.7%
18x22	95.9%	88.1%	95.7%	96.8%	91.5%	96.7%	94.5%	84.2%	94.3%	95.4%	91.5%	95.4%
19x22	95.4%	90.9%	95.2%	98.4%	85.6%	97.9%	93.1%	84.4%	92.8%	95.6%	84.6%	95.2%
20x22	95.2%	90.0%	94.8%	96.3%	89.5%	95.8%	97.9%	83.8%	96.9%	95.3%	90.5%	95.0%
21x22	98.5%	86.9%	98.2%	95.6%	88.9%	95.4%	97.0%	90.4%	96.9%	98.5%	89.4%	98.3%
22x22	96.0%	87.2%	95.6%	96.0%	87.8%	95.6%	95.4%	83.5%	94.9%	97.5%	86.9%	97.1%
23x22	96.2%	85.2%	96.0%	98.2%	88.9%	98.0%	98.5%	90.4%	98.4%	98.7%	88.1%	98.5%
24x22	95.0%	85.9%	94.7%	96.8%	85.3%	96.4%	97.6%	87.7%	97.3%	96.0%	90.2%	95.8%
25x22	98.6%	91.8%	98.5%	96.9%	89.0%	96.7%	97.8%	89.0%	97.6%	96.3%	89.0%	96.2%
26x22	97.1%	88.4%	96.9%	96.2%	88.4%	96.0%	94.9%	83.7%	94.7%	96.1%	90.7%	96.0%
27x22	97.1%	92.2%	96.7%	98.0%	90.2%	97.4%	97.5%	87.3%	96.8%	98.1%	92.2%	97.7%
28x22	96.2%	92.9%	96.0%	96.0%	85.7%	95.4%	98.7%	89.3%	98.2%	95.4%	89.3%	95.0%

Table 11: Comparison of Loss between Mutation and GAN Balanced Datasets

Size	Mutation-balanced Training Datasets						GAN-balanced Training Datasets					
	5x5 DNN			10x10 DNN			5x5 DNN			10x10 DNN		
	SAT	UNSAT	Overall	SAT	UNSAT	Overall	SAT	UNSAT	Overall	SAT	UNSAT	Overall
11x22	0.0447	0.1103	0.0453	0.0410	0.0239	0.0319	0.0510	0.1021	0.1013	0.0841	0.0372	0.0751
12x22	0.0013	0.0681	0.0629	0.0582	0.0448	0.0488	0.0523	0.0512	0.0514	0.0896	0.0877	0.0882
13x22	0.0769	0.0218	0.0373	0.0708	0.1175	0.1149	0.0672	0.1034	0.0785	0.1014	0.0584	0.0990
14x22	0.0392	0.1029	0.0458	0.0789	0.0189	0.0251	0.0104	0.0239	0.0133	0.0587	0.0094	0.0398
15x22	0.0191	0.1197	0.1057	0.1008	0.1096	0.1056	0.0604	0.1169	0.0614	0.0412	0.0795	0.0767
16x22	0.0761	0.0199	0.0252	0.0906	0.0691	0.0698	0.0073	0.0784	0.0505	0.0618	0.0220	0.0285
17x22	0.0119	0.0249	0.0164	0.0727	0.0769	0.0738	0.0162	0.0217	0.0193	0.0713	0.0429	0.0621
18x22	0.0380	0.0651	0.0418	0.0044	0.0414	0.0330	0.0128	0.0081	0.0108	0.0910	0.0475	0.0896
19x22	0.0333	0.0264	0.0306	0.0347	0.0566	0.0386	0.0570	0.0277	0.0483	0.0504	0.0721	0.0710
20x22	0.1021	0.0136	0.0189	0.0435	0.1051	0.0799	0.1167	0.1018	0.1069	0.0847	0.0352	0.0379
21x22	0.0204	0.0134	0.0148	0.0624	0.0304	0.0589	0.0757	0.1035	0.0938	0.1085	0.0663	0.1016
22x22	0.1172	0.1019	0.1128	0.0289	0.0636	0.0479	0.0456	0.0528	0.0511	0.0364	0.0549	0.0545
23x22	0.1195	0.0910	0.0911	0.0781	0.0748	0.0759	0.0981	0.0087	0.0526	0.0847	0.0216	0.0348
24x22	0.0405	0.0172	0.0264	0.0993	0.0986	0.0987	0.0336	0.0327	0.0335	0.0875	0.1087	0.0904
25x22	0.0270	0.0083	0.0106	0.0483	0.0478	0.0478	0.1116	0.1031	0.1071	0.0520	0.0964	0.0546
26x22	0.0123	0.0757	0.0429	0.0440	0.1154	0.0765	0.0717	0.0263	0.0603	0.0211	0.0130	0.0157
27x22	0.0565	0.0916	0.0730	0.0635	0.0031	0.0537	0.0819	0.0058	0.0618	0.0778	0.0308	0.0570
28x22	0.0239	0.0741	0.0427	0.0149	0.1091	0.0169	0.0653	0.1154	0.0893	0.0417	0.0496	0.0452

However, compared to directly mutating existing path conditions, training a GAN is far more expensive in time cost. Since the matrix sizes are different, for each DNN we need to train a GAN, which usually takes more than 20 hours before we can generate new path conditions, while mutation can simply be applied on all matrices with different sizes. Meanwhile, different from common GAN on images, we do not have a direct yet objective method to evaluate the generated path conditions. In other words, it is hard to tell the quality of generator itself. More importantly, theoretically GAN only generates data that are similar to the samples in the original dataset, which does not necessarily increase the diversity of the training dataset. As we see from Table 10 and Table 11, using a GAN does not lead to a higher accuracy on classifying UNSAT path conditions. In contrary, for many cases the accuracy dropped compared to using the mutation-based

balancing method while the data loss remains low. Thus, mutation-based balancing method is more cost-effective, and we use DNNs trained with mutation-based training datasets for the rest of our evaluation.

- *RQ3: How efficient is DeepSolver in classifying path conditions?*

We performed satisfiability checking using Z3, Green, GreenTrie, and DeepSolver on the same set of path conditions as collected in the experiment for RQ1. For Green and GreenTrie, all the balanced training data (1, 016, 524 constraint solutions) are used for their solution store, and Z3 is invoked when a cache miss happens.

Table 9 shows the average time cost of each technique for satisfiability checking for path conditions sized from 11×22 to 28×22 for each subject. The number of path conditions in each subject range from 716 to 23, 175. Our observation is that DeepSolver (both 5×5 and 10×10) outperforms Green and GreenTrie for all subjects, while all three techniques are faster than Z3 as expected. Consider MergeSort as an example, Z3 took 98.18 milliseconds on average, Green took 70.88 milliseconds (1.39X faster than Z3), GreenTrie took 66.76 milliseconds (1.47X faster than Z3), and DeepSolver with 5×5 DNNs and DeepSolver with 10×10 DNNs took 3.63 milliseconds (27.04X faster than Z3) and 6.72 milliseconds (14.62X faster than Z3), respectively. Moreover, the cost of DeepSolver is consistently low across different subjects. Particularly, DeepSolver (5×5) spends 3-4 milliseconds in classifying a path condition. In contrast, the performance of Green and GreenTrie is not consistent because they highly depend on the reuse rate, as they still need to invoke Z3 when there is no matching of record for reuse.

- *RQ4: How does the number of hidden layers/neurons impact DeepSolver's efficacy in classifying path conditions?*

We use the results from the experiments for RQ1 and RQ3 to answer this research question.

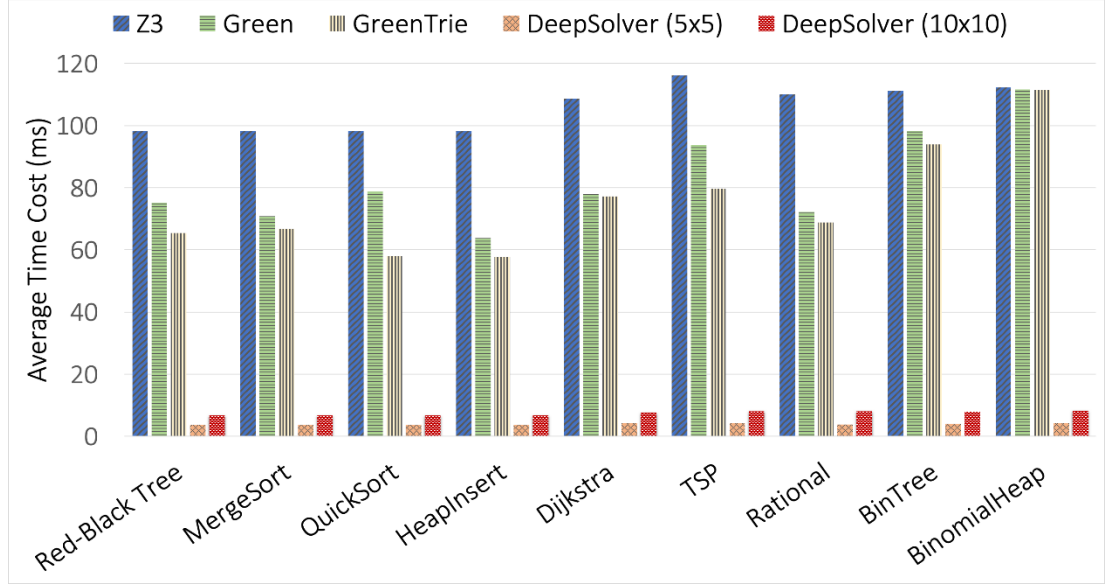


Figure 11: Average time cost (ms) of satisfiability checking

Table 8 shows the results of DeepSolver, when the two configurations of hidden layers/neurons (i.e., 5×5 and 10×10) were used. For each size of the path conditions, we highlighted the cells in the table where DeepSolver (10×10) achieved higher accuracy than DeepSolver (5×5). We find exactly half of the SAT cells, half of the UNSAT cells, and half of the Overall cells are highlighted. Moreover, the difference in accuracy achieved by the two DNN configurations is small. For example, for path conditions sized 22×22 , DNN (10×10) achieved slightly higher accuracy than DNN (5×5), while for path conditions sized 26×22 it is just the opposite. The results indicate that the number of hidden layers/neurons does not have significant impact on the

accuracy of DeepSolver. For reference, DNNs trained by GAN-balanced datasets are also reported and highlighted in Table 9. We will not expand the discussion as no significant difference to data in Table 8 is observed. On the other hand, according to the efficiency results shown in Figure 11, DeepSolver (10×10) consistently cost about double the time of DeepSolver (5×5) across all the subjects, which indicates that the configurations of hidden layers/neurons do have an impact on the efficiency of DeepSolver. This is expected since the input data must go through more layers and neurons in a larger DNN mode.

- *RQ5: How efficient is symbolic execution with DeepSolver?*

According to the results for RQ3, GreenTrie is more efficient than Green and Z3, so we compare DeepSolver with only GreenTrie to answer this research question.

We ran SPF with DeepSolver (Algorithm 5) on the nine subjects that were not used for training DNNs. Z3 was used to solve path conditions, when they are not supported by the DNNs that have been trained, or when they need to be solved by a conventional constraint solver (for double-check and test generation). For comparison, we ran SPF with GreenTrie on the same subjects, and Z3 was invoked when a cache miss happens. All symbolic execution runs were configured with a time bound of 5 hours and a depth bound of 28, since we have no DNNs trained for path conditions with more than 28 constraints. Please note this is not a limitation of DeepSolver. In fact, DeepSolver can be used to classify path conditions with any number of symbolic variables and constraints, as long as we have enough data to train the DNNs.

Table 12: Results of symbolic execution with DeepSolver versus GreenTrie

Subject	SPF with GreenTrie			SPF with DeepSolver					
	# PC	# State	Time Cost (s)	# PC	SAT-To-UNSAT Misclassification	UNSAT-To-SAT Misclassification	# State	# Leaf State	Time Cost (s)
Red-Black Tree	1,329	1,330	461	1,342	41	13	1,343	107	216
MergeSort	8,044	8,045	2,243	8,105	197	61	8,106	825	1,963
QuickSort	23,209	23,210	4,193	23,271	730	62	23,272	984	2,599
HeapInsert	2,580	2,581	683	2,589	65	9	2,590	115	525
Dijkstra	10,646	10,647	1,823	10,706	400	60	10,707	526	1,371
TSP	13,212	13,213	2,422	13,276	350	64	13,277	857	1,262
Rational	744	745	303	745	28	1	746	15	137
BinTree	3,467	3,468	1,036	3,480	85	13	3,481	119	390
BinomialHeap	23,216	23,217	8,164	23,304	76	88	23,305	472	3,192

Table 12 shows the results. For both techniques, we report the number of path conditions (# PC) that were analyzed, the number of states explored by SPF (# State), and the total time cost. For SPF with DeepSolver we also report two types of misclassification errors and the number of leaf states. We observe that number of misclassification errors is relatively small compared to the total number of path conditions, due to the high accuracy of DeepSolver. Also because of UNSAT-To-SAT Misclassification, SPF with DeepSolver analyzed slightly more path conditions and thereby explored more states than SPF with GreenTrie. Despite the overhead introduced to address the two types of misclassification, we find that SPF with DeepSolver is still faster than SPF with GreenTrie across all the nine subjects. For example, SPF with DeepSolver achieved 2.56X speedup on BinomialHeap). Please note that we used Z3 to solve path conditions sized from 1×22 to 10×22 . If we have enough constraint solutions to train DNNs to support these path conditions, the performance of DeepSolver could be even better. The results demonstrate that our strategies proposed in Section V

are effective in addressing the two types of misclassification, and DeepSolver can enable more efficient symbolic execution while ensuring its soundness.

Discussions

Threats to Validity

The primary threat to external validity in this study involves the subjects selected for our study, the constraint solutions collected for training DNNs, deep learning algorithm and the two DNN structures used in our study. Although we used the subjects that were widely used in the literature, the results may not generalize to other subjects. While mechanical transformation can generate more constraint solutions and address class imbalance problem, the experimental results suggest that constraint solutions generated in such way may not be as diverse as those collected from real programs and may decrease the accuracy of DeepSolver. These threats need to be further addressed by additional studies on different subjects, with more real constraint solutions, using other deep learning algorithms and other DNN structures.

The primary threat to internal validity lies in the possible errors in our implementation and the tools used in our study including Keras, TensorFlow, and SPF. We controlled for this threat through the use of extensive functional tests on our implementation and these tools and verification against cases in which we can manually determine correct results. Meanwhile, over-fitting is a common problem when training DNNs. The experimental results show that all DNNs used in the study have stable and high accuracy on different datasets, and there seems no over-fitting problem with these DNNs.

Limitations

While DeepSolver can classify path conditions for satisfiability, it does not provide a solution for satisfiable path condition like conventional constraint solvers. When a solution is required, e.g., for generating test cases, a conventional constraint solver would be required. In addition, our work currently supports for classification of linear integer arithmetic path conditions. Choosing informative, discriminating, and independent features is a crucial step for effective algorithms in pattern recognition and classification. For example, the features for computer vision are different from features for speech recognition. Similarly, path conditions in other theories (e.g., theories of bit vectors, strings, and arrays) may expose different features, and thus require different algorithms to learn those features. We would like to extend the range of theories supported by DeepSolver for our future work.

Comparison with Random Approach

Given that the majority of the path conditions in symbolic execution are satisfiable, a random approach without the trained DNNs may be believed to outperform DeepSolver. More specifically, such random approach would randomly guess the satisfiability according to the percentage of satisfiable path conditions. For example, if 90% path conditions are satisfiable for a program, then the random approach would blindly guess a path condition is satisfiable with a 90% possibility. While the strategies we proposed in Section V may still work for random approach to deal with the two types of misclassification, we would like to argue that random approach is not applicable in practice. First of all, there is no prior knowledge on the percentage of satisfiable/unsatisfiable path conditions of a program. Second, even such knowledge is

available and the random approach may provide a high accuracy for classifying satisfiable path condition, it would lead to a low accuracy for classifying unsatisfiable path conditions, which makes symbolic execution to explore many infeasible states.

To confirm our understanding, we implemented such random approach and evaluated on the nine subjects listed in Table 12. We ran symbolic execution with this random approach using the same time bound and depth bound as in RQ4. For each subject, we ran 10 times. The results indicate that many symbolic execution runs with the random approach timed out, and when it did not time out, the time cost is still higher than DeepSolver. For example, for BinomialHeap, six out of 10 runs timed out, one run took 8,305 seconds, two runs took around 5,500 seconds, and one run took 4,162 seconds, compared to 3,192 seconds taken by DeepSolver.

VI. CONCRETIZATION FOR CONSTRAINT ANALYSIS

This chapter introduces Cocoa, an optimized constraint solving solution by reducing the complicity of PC with concretization. We will use the example of running symbolic execution on DNNs as a typical scenario to describe our approach in this chapter, as DNNs turned to generate complex PC that could not be solved in a short time.

Overview

Being a cutting-edge technique, Deep Neural Networks (DNN) are increasingly and widely developed and used in both research area and industry. One typical example is DNNs are used for image classifying tasks. It takes in an image or video, which is usually considered a high dimensional and complex matrix input and calculates and transforms the input data through neurons on multiple layers, which finally assign the input to a predefined output label. Such DNNs are integrated into the structure of auto driving vehicles to help the system decide its behavior, and multiple major car companies such as Tesla and BMW are putting their resources on developing them. Obviously, such applications need to be carefully tested before deployment, for a wrong decision-making may lead to great economic loss or even fatal incidence. However, many of the DNN applications are with substantial safety and security concerns (LeCun et al., 2015), for thoroughly testing a deep neural network remains a difficult task. Validating DNNs is complex and challenging, due to the nature of the learning techniques that create these models. Although it is possible to convert a DNN implementation into a traditional software with relatively simple source code, it is challenging applying conventional techniques such as symbolic execution on it directly.

We use an example of a DNN trained on learning library of MNIST (mni, 2019) to explain our insight. MNIST is a database of handwritten digits that is widely used for evaluating deep learning techniques. Each digit is a gray-scale image of the size 28x28. A sample data from MNIST is shown in Figure 12. Typically, a deep neural network treats each pixel of the image as an independent input variable, and each neuron in the deep neural network performs a calculation based on these values and then compares the result with a threshold. The calculation accumulates and propagates till the output layer, and the final calculation result is used to make a decision.

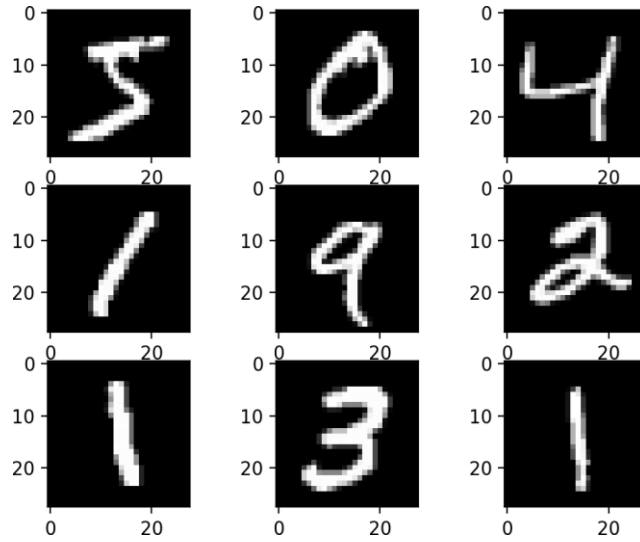


Figure 12: Sample data of MNIST database

Obviously, since this accumulation of calculation and comparison are very similar to symbolic execution, we can apply symbolic execution on a DNN to generate test cases that covers a new path of classification. However, in practice, we noticed that even though the DNN is usually simple in the shape of conventional programming source code and the state space is easy to explore, solving a PC generated by DNN can be extremely slow due to following reasons:

- Large number of symbolic variables. Since each pixel is considered a different input parameter for a neural network, this method will have 784 inputs and one return value, indicating the classification result. As we can see, comparing to a common method in traditional programs, the number of input of parameters is much larger. If the neural network is using a fully connected structure, each constraint will involve all symbolic variables. It obviously increases the complexity of the whole path condition.
- Extra restriction of symbolic variables. Since each input parameter represents a pixel in the image, we need to put a data range for each of them. The original data range for a pixel in gray-scale image is an integer between 0 and 255. Meanwhile, normalization is a widely used technique in image processing and analysis (Finlayson et al., 1998), so that the data for a deep neural network is commonly not as integer values. For instance, a gray-scale image is commonly normalized into a range of $[0, 1]$. If we take this factor into consideration in symbolic execution, we need to treat the path constraints as a calculation between real numbers, which further increased the difficulty of finding a potential solution.
- Large number of constraints. Generally speaking, a deep neural network is commonly built by multiple dense layers, which means each neuron in a layer is fully connected to all the neurons in its previous and following layer. Consider a neural network with 10 neurons in each layer to classify a MNIST image. If we look at a neuron in the second hidden layer, it takes the output from all 10 neurons in the first hidden layer. Remember that each of the neurons in the first layer represents a constraint of 784 symbolic variables. Thus, a path condition towards a

neuron in second layer should be consisted of 10 such constraints. We can see that as the neural network goes deeper, the complexity of path condition increases dramatically.

In order to make symbolic execution applicable on subject with complex PCs such as DNNs, we need to reduce the time cost of solving complex PCs. Our solution is to concretize a PC by replacing part of the symbolic variables with their concrete values. This solution is partially inspired by the concept of concolic execution (Kim et al., 2012a), in which the concrete input value is used to direct symbolic execution in order to improve its efficiency to reach a high coverage. Unlike in normal concolic execution where concrete values of all inputs are used to discover new paths in the state space, we only use part of the data to concretize the least “important” symbolic variables in the PC. We still take the DNN for MNIST image classification as an example. Our insight is that not all pixels are equally important for an image classification task. For MNIST database, each image can be considered containing both a “background” (the dark pixels) and the “actual data” (the light pixels). Ideally, we can skip or ignore the background part of image when classifying the number, as the pixels in background do not have any interesting features. In other words, we can assume at since the unimportant pixels contribute less on decision-making, in a new path it would be safe to keep them as their original concrete value, and we are more interested in whether changing only the most important input variables can change the output of a program. Concretization may not only reduce the number of symbolic variables but also the number of constraints, as after the replacement it is possible to have some constraints converted into a simple comparison between constant values, which is fast to justify. Nevertheless, we expect the

concretized PC to be less complex than the original one and thus we need less time to solve it. We need to mention that in this dissertation, we only use DNNs as example for it is a typical type of programs that can generate a PC that cannot be solved in a short time. However, the insight of optimizing constraint solving by concretization can be applied on any subject that has complex PCs.

Cocoa

Symbolic Execution on DNNs

In order to apply symbolic execution on DNNs, the first step is to convert the DNN into an imperative code. By definition, a neuron in DNN takes the output from other neurons (in most cases, the neurons from its previous layer) as an input matrix. These input matrices are then processed and calculated, and a threshold will be used to decide the output. This process is usually referred as “activation functions” (Nwankpa et al., 2018). One typical activation function is rectified linear units, or ReLUs (Nair and Hinton, 2010), which is widely used for it is biologically plausible (Glorot et al., 2011).

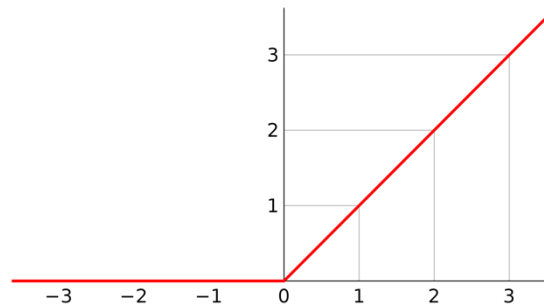


Figure 13: Example Plot of ReLUs

Figure 13 shows a plot of ReLUs. In DNNs, ReLUs can be expressed as $f(x) = \max(0, WT \times x + b)$, where x represents the input matrix, WT represents the weight matrix and b represents the bias. Calculation result is only used when it is greater than a

threshold (which is 0 in our example), or else the neuron will output the threshold instead, which is also referred as “inactivated.”

Based on the fact that each neuron in DNN represents an activation function, it is possible to convert a neuron in DNN model into a method in conventional programs, while each method has a branch to decide the output. A DNN model is then transformed into a chain of method calls, where classifying one input can be considered a certain execution path in the program. As a result, symbolic execution can then be applied on the converted program as we try to generate inputs for different classification paths. In our work, we apply the technique of DeepCheck (Gopinath et al., 2018) for DNN conversion. For the rest of our paper, we note a program converted from DNN as “DNN program” to better explain the technique and evaluation.

Constraint Concretization

Our solution aims to speed up the constraint solving by concretizing the PC generated from a DNN program. Similar to concolic execution (Kim et al., 2012a), we first apply a concrete execution by running the DNN program with an input and collect the constraints following the path. Then, we flip one of the constraints and apply symbolic execution on the corresponding path, so we can collect the full PC of it to generate the input. However, instead of solving the original PC with all symbolic variables as in normal concolic execution, we concretize part of the symbolic variables with their concrete value from original input.

Ranking Heuristics

While replacing a symbolic variable is technically easy, how to choose symbolic variables to replace remains worthy of being discussed. The most straightforward way is

to randomly pick up a subset of variables, and this solution can be applied in traditional programs for the variables can be considered equally important. However, it is too naive as a solution to apply on DNN programs. In practice, it's common that the symbolic variables are in fact not equally important. Since artificial neural network can be considered inspired by biology nervous system (Zhang, 2000), we can use a classic psychology problem called “cocktail party effect” (Bronkhorst, 2015) to reason such an observation. This effect shows that there is a mechanism in our nervous system that can pick up and process only the information we are more interested in from a noisy background. Similarly, in our DNN program example, although DNN reads in all pixels in the image, not all of them are “interesting.” On the contrary, the pixels or areas containing the features to make classification decision is actually limited. As a result, a more reasonable way to choose symbolic variables for concretization is to rank the symbolic variables based on an importance score. If a symbolic variable has a high importance score, it represents certain features for decision-making. Thus, we can keep it as symbolic, and only replace the least important pixels with their concrete values.

In our work, we introduce and evaluate 5 different ranking heuristics.

- *COE*: We consider the actual values of the coefficients to determine the impact of the input variables on decision-making. Defining a PC as $y_1 \wedge y_2 \wedge \dots \wedge y_n$, where $y_n = c_1^n \times x_1 + c_2^n \times x_2 + \dots + c_k^n \times x_k + b \text{ op } 0$, for each symbolic variable x_k , we can calculate its importance score Imp_k as $c_k^1 + c_k^2 + \dots + c_k^n$. We maintain a list of symbolic variables sorted based on their importance scores in ascending order, for a symbolic variable with less coefficient can be considered contributing less to calculation and final decision-making.

- *ABS*: One potential weakness of *COE* is that it could be difficult to argue whether simply sum up the value of coefficient of a variable can reflect its importance in decision-making. Consider a simple path constraint $2 \times x_1 + x_2 = 0 \wedge -1 \times x_1 = 0$ as example. With *COE*, both x_1 and x_2 have the same importance score of 1, which is counter-intuitive for while x_1 participated in two constraints, x_2 only appeared once. To reflex this problem, we introduce our second ranking heuristic *ABS*, which sum up the absolute value of coefficients for a variable one calculating the importance score: $Imp_k = |c_k^1| + |c_k^2| + \dots + |c_k^n|$.
- *COE-VAL*: Result of an active function depends not only on the weight of each variable, but the value of it as well. In other words, in order to identify pixels that can be attributed for the classification decision for the specific given input, we need to consider the input values as well. This can be determined by multiplying the coefficient values with the corresponding input values, similar to techniques such as DeepLIFT (Shrikumar et al., 2016). The importance score can be noted as $Imp_k = (c_k^1 + c_k^2 + \dots + c_k^n) \times v_k$, where v_k represents the concrete value of variable x_k .
- *ABS-VAL*: Similar to *COE-VAL*, we consider both the coefficient and value of a variable to calculate the importance score, while in this strategy we use the absolute value of coefficients as in *ABS* instead of true value. The importance score can be noted as $Imp_k = (|c_k^1| + |c_k^2| + \dots + |c_k^n|) \times v_k$.
- *STEP*: The last ranking heuristic is designed to support concolic execution. Different from the previous strategies, in *STEP* we cannot simply calculate the importance score for ranking. In concolic execution, a common way to decide the

next path to explore is by negating the last constraint in PC. Thus, the variables in the last constraint can be considered most important. However, since a symbolic variable usually appears in multiple constraints, it is possible for other symbolic variables to participate the calculation as well. These symbolic variables who participated in the calculation with important symbolic variables can also be considered as important. That is to say, the number of important variables can be decided as we update the list in multiple iterations. In our heuristic, we use a constant S to decide the number of iterations. When S is set to 0, we add only the symbolic variables in the last constraint to the list of important symbolic variables. As the value of S increases, we update the list with the other variables which participated in calculation with the variables that are already on the list.

Consider an example PC of $a + b + c = 0 \wedge b + c + d = 0 \wedge d + e = 0 \wedge e + f = 0$. In this example, we have six symbolic variables (a to f) in total. If we apply Step with $S = 0$, we only consider the variables appeared in the last constraint, which are e and f , and put them in the list of important variables we would like to keep symbolic. If $S = 1$, we apply one iteration of updating the list, in which we find variable d participated in calculation with variable e , so we add variable d to the list as well. Similarly, if $S = 2$, variable b and c will also be added to the list, in which case only symbolic variable a will be replaced by its concrete value.

Noted that these heuristics we introduced in this work is typically designed for symbolic execution on our DNN program of MNIST database, which is used in our case study shown later. It is possible that other ranking heuristic can be applied for different

DNN model. In this dissertation, these five heuristics are chosen only to better illustrate our technique.

Implementation

We use two algorithms to show the detail of our implementation. First, Algorithm 6 shows the process of calculating the rankings that is used for *COE*, *ABS*, *COE-VAL*, and *ABS-VAL*. Note that this algorithm is only for the convenience of description. Note that this algorithm is only for the convenience of description. In actual implementation, we do not need to calculate all four ranking scores at the same time. We consider the four heuristics are combinations of two configurations, which are whether we should use absolute value of coefficients, and whether we should consider the concrete value of symbolic variables. We use two Boolean values, namely *ABS* and *VAL*, to represent these two settings. After initializing the coefficient map *Map* and return list *RankList* (Lines 1-2), we first break down the input path condition *PC* into constraints and then into the calculation between symbolic variables $co \times sym$ (Lines 3-4). We update the coefficient map using value of coefficient *co* based on *ABS* : if *ABS* is false, we update *Map* with the actual value of *co*, which can be used for *COE* and *COE-VAL* (Lines 5-11). Otherwise, we update *Map* with the absolute value of *co*, which can be used for *ABS* and *ABS-VAL* (Lines 12-17). After checking all the calculations on symbolic variables in *PC*, *Map* contains the information of the symbolic variables with their importance based on coefficients, where the keys in *Map* are symbolic variables (noted as *sym*) and the values are the scores (noted as *score*). If *VAL* is false (Line 22), we do not consider the concrete value of symbolic variables and we put all the key-value pair in *Map* into *RankList* in shape of $(score, sym)$ directly (Lines 23 - 25). Otherwise, we get the concrete value for

each symbolic variable *sym* from the concrete value map *ConMap*, and use the concrete value *con* as well as the importance from Map to update *RankList* with $(score \times con, sym)$ (Lines 26-31). Finally, we sort *RankList* in ascending order based on score (Line 32) so that the least important symbolic variables appear at the front of the list. After we return this list (Line 33), we can use different strategies to decide the portion of the list for concretization such as using fixed numbers or a percentage.

Algorithm 6 Algorithm of Ranking Heuristics

Input: Path constraint PC , Absolute mode Boolean ABS , Value mode Boolean VAL , Map of concrete value $ConMap$

Output: Ranking List of symbolic variables $RankList$

```
1: Map of coefficient  $Map \leftarrow null$ 
2:  $RankList \leftarrow null$ 
3: for all Constraint  $c$  in  $PC$  do
4:   for all  $co \times sym$  in  $c$  do
5:     if  $ABS$  is false then
6:       if  $Map.contains(sym)$  then
7:          $Map.put(sym; co)$ 
8:       else
9:          $preval \leftarrow Map.get(sym)$ 
10:         $Map.put(sym, co + preval)$ 
11:      end if
12:    else
13:      if  $Map.contains(sym)$  then
14:         $Map.put(sym, |co|)$ 
15:      else
16:         $preval \leftarrow MAP.get(sym)$ 
17:         $Map.put(sym, |co| + preval)$ 
18:      end if
19:    end if
20:  end for
21: end for
22: if  $VAL$  is false then
23:   for all  $(sym, score)$  in  $Map$  do
24:      $RankList.put(score; sym)$ 
25:   end for
26: else
27:   for all  $(sym, score)$  in  $Map$  do
28:      $con \leftarrow ConMap.get(sym)$ 
29:      $RankList.put(score \times co, sym)$ 
30:   end for
31: end if
32:  $RankList.sort\_asc()$ 
33: return  $RankList$ 
```

Algorithm 7 shows the algorithm of heuristic STEP. We do not rank symbolic variables based on an importance score, but we need to iteratively update a list of important variables that we want to keep as symbolic. The number of iterations as an input is noted as S . First, we generate two list of symbolic variables: one to keep all the

important variables noted as *ImportantVarList* (Line 1), and one to keep the symbolic variables we want to concretize noted as *ReplaceList* (Line 2). Since the algorithm is similar to traversing a map with multiple start points, where each symbolic variable is a node and the relationship between symbolic variables is an edge. We need to maintain a map of two-way “edges” noted as *Relations* (Line 3). We check all constraints in *PC* in order, and for each pair of symbolic variables (sym, sm') appeared in the constraint, we add both (sym, sm') and (sm', sym) into *Relations* to show that they are directly related to each other (Lines 4-7). Meanwhile, for all the symbolic variables we encountered, we add it to the *ReplaceList* (Line 8-9). As we reached the last constraint in *PC*, we add all symbolic variables in that constraint to *ImportantVarList* to create the start point for traversing (Lines 10-14). After the map is generated, we recursively update *ImportantVarList* on the value of *S* (Line 16). In each iteration, we create a temporary list to store the symbolic variables to be updated (Line 17). We check all the symbolic variables in current *ImportantVarList*, and put all the “nodes” that can be reached with one step into the temporary list (Lines 18-20). Then, we delete the two-way edge from relationship map to speed up further iterations (Lines 21-24). Once *ImportantVarList* is fully processed, we update it with the symbolic variables in the temporary list (Line 25) and decrease the value of *S* by one before entering the next iteration (Lines 26-27). When *S* reduced to 0, the list of important symbolic variables is fully updated. Then, we remove all the symbolic variable in this list from the *RelaceList*, and return *ReplaceList* (Lines 28-31). Once *ReplaceList* is ready, a path constraint can be simplified by replacing the symbolic variables in it (either a portion of *RankList* in Algorithm 6 or the *ReplaceList* in Algorithm 7) with provided concrete value map, as

ConMap we used in Algorithm 6. For our task, this map is easy to get by storing the actual values used in actual classification, while the coordinates of pixels are used to generate the names of symbolic variable.

Algorithm 7 Algorithm of *STEP* Heuristic

Input: Path constraint *PC*, Number of steps *S*

Output: List of symbolic variables to be replaced *ReplaceList*

```

1: ImportantVarList  $\leftarrow$  null
2: ReplaceList  $\leftarrow$  null
3: Map of symbolic variable pairs Relations  $\leftarrow$  null
4: for all Constraint c in PC do
5:     for all symbolic variable pair (sym, sym') in c do
6:         Relations.put(sym, sym')
7:         Relations.put(sym', sym)
8:         ReplaceList  $\leftarrow$  sym
9:     end for
10:    if c is the last constraint in PC then
11:        for all symbolic variable sym in c do
12:            ImportantVarList.put(sym)
13:        end for
14:    end if
15: end for
16: while S > 0 do
17:     Temporary list temp  $\leftarrow$  null
18:     for all symbolic variable sym in ImportantVarList do
19:         for all (sym, sym') in Relations do
20:             temp.put(sym')
21:             Relations.put(sym, sym')
22:             Relations.put(sym', sym)
23:         end for
24:     end for
25:     ImportantVarList.put(temp)
26:     S  $\leftarrow$  S - 1
27: end while
28: for all symbolic variable sym in ImportantVarList do
29:     ReplaceList.remove(sym)
30: end for
31: return ReplaceList

```

Case Study

Before we evaluate our solution, we first conduct a case study on two questions.

- **Case study question 1:** Among all 5 heuristics, which one is more promising to be applied in evaluation?
- **Case study question 2:** What is the performance we can expect from solving a concretized PC?

Experiment Setup

We implement a fully connected DNN with the neuron structure of $784 \times 10 \times 10 \times 10$ (784 inputs, 10 hidden layers with 10 neurons in each layer, and 10 classification results). The DNN is trained on all 60,000 images in the training data of the MNIST dataset and reached an accuracy of 94%. We converted this DNN into a program, and applied Badger (Noller et al., 2018), a fuzzing and symbolic execution framework, to generate PCs. Badger is executed on the DNN program with an image of digit 6 (Figure 14) as the initial concrete input. Then, we ran the fuzzing for 24 hours and generated a collection of 1,216 satisfiable PCs in total.



Figure 14: Example Image from MNIST



















































Case Study on Heuristics

For this case study, we use the PC for correctly classifying the input (Figure 14) to check the five heuristics. For heuristics COE, ABS, COE-VAL and ABS-VAL, we

setup the percentage of important pixels from 50% to 0% with pace of 5%, which represent 11 decisions ranging from keeping half of the pixels in the original image as symbolic to not to concretize any of the pixels. Accordingly, we run the STEP heuristic with the value of S set from 0-9, representing only keep the pixels in the last constraint in each PC to running up to 9 iterations of updating the list of replacement. We run each heuristic with different configurations and mark the most important pixels for concretization in red color for manually comparison. Noted that these red pixels are supposed to be left as symbolic in concretization. The output images for each configuration are shown in Table 13. Based on the results, we have the following two observations:

First, for our DNN program, heuristic STEP appears to be the least reasonable. As we can see from the result images, this heuristic failed to reflect or capture logically important partition of the input image. The highlighted pixels (or input variables) expand from the left top corner, which is black background without any feature. We consider this a result caused by the design of our DNN: Since the task is a typical image processing, we introduced a convolutional layer, which uses a 3×3 convolution kernel, as the first hidden layer. As a result, the last constraint in PC always represents the first window with 9 pixels for calculation. We find that for a convolution kernel with the size of $K \times K$, the number of important pixels shows a cubic growth of $N = (K + 2 \times S)^2$ as the value of S increases. This shows another weakness of this heuristic that we cannot obtain a finer control of the number of symbolic variables to be kept, for we cannot manually decide the size of list holding important variables as it is updated based on a strict logic on the relationship between variables.

Table 13: Result of Ranking Heuristics

Ratio of Important Variables	COE	ABS	COE-VAL	ABS-VAL	Iterations	STEP
50%					9	
45%					8	
40%					7	
35%					6	
30%					5	
25%					4	
20%					3	
15%					2	
10%					1	
5%					0	

Second, comparing with a ranking heuristic purely based on coefficients (COE and ABS), considering the concrete values with coefficients (COE-VAL and ABS-VAL) is more powerful in discovering the logical important part of the image. While the important pixels are relatively evenly distributed in COE and ABS, COE-VAL and ABS-VAL concentrate more on the parts that has some interesting features i.e. the “circle” of digit 6. Meanwhile, ABS-VAL turns to have a faster discovery of the features. When we set the ratio of replacement to 25%, ABS-VAL concentrates the most on the circle part of digit 6 comparing with all other heuristics, while COE-VAL can only reach a similar

discovery with a ratio of 30%. This result shows that ABS-VAL heuristic can identify important pixels that are closest to manual judgment.

Based on these observations, we choose ABS-VAL as our heuristic for the rest of evaluation. However, we also need to mention that our ABS-VAL algorithm is typically designed for a DNN trained on gray-scale images with black background. Should the characteristic of input data be changed (e.g. inverted into white background with black digits), the heuristic needs to be updated accordingly. It is not always easy to find a reasonable way to calculate the coefficient with the value. For instance, a colored image usually has a higher dimension (e.g. RGB images has at least three layers while a gray-scale images only have one). This can lead to the problem of “curse of dimensionality” (Keogh and Mueen, 2017). That is to say, among all the dimensions of the input data, it is difficult to decide the importance of each dimension, and more difficult to find a reasonable way to express it.

Case Study on Performance

For our case study, we concretize the 1,216 PCs generated by Badger from the DNN program. Each PC is concretized with ABS-VAL heuristic and a concretization ratio ranges from 50% to 0%. Please be noted that this ratio has a different meaning than in the last case study: in Table 13, a ratio of 5% means we consider 5% of the pixels as important, so we will replace 95% of symbolic variables with concrete values, while in this case study (as well as the rest of this dissertation), a 5% ratio means we would like to concretize 5% of symbolic variables.

We apply two widely used Satisfiability Modulo Theories (SMT) solver Z3 (Z3S, 2019) and CVC4 (CVC, 2020) to solve all PCs before and after concretization with a two hours' time limit.

Results and Analysis

The performance is shown in Table 14 and Table 15. We group the results into three categories based on the result from the solver, namely the PC is satisfiable (SAT), unsatisfiable (UNSAT) or cannot be solved within the time limit (TimeOut) and report their number for each ratio. For SAT and UNSAT PCs, we also report the number of concretized PCs that took more time to solve comparing with original PC (Longer) and the number of those took less time (Shorter), and calculate the possibility of getting a speedup as the percentage of Shorter among the sum of Longer and Shorter PCs in the same category. Meanwhile, we also report the accuracy for simplified PCs calculated as the percentage of SAT concretized PCs among all the 1,216 generate PC for each ratio.

To have a better look at the distribution of the performance, we further group the result into a figure as shown in Figure 15. For all concretized PCs result in SAT or UNSAT, we calculate the percentage of the time cost of solving the corresponding original PC. Furthermore, each concretized PC is categorized based on the solver and the result as shown in this bar chart.

Table 14: Performance of Z3 Solving Simplified PC

Z3												
Ratio of Concretization	Total	50%	45%	40%	35%	30%	25%	20%	15%	10%	5%	0%
SAT	Longer	3,055	48	86	110	116	127	125	178	367	644	624
	Shorter	3,013	93	133	149	146	151	166	191	299	509	592
	Total	6,068	141	219	259	262	278	291	369	666	1,153	1,216
	Speedup Possibility	49.65%	65.96%	60.73%	57.53%	55.73%	54.32%	57.04%	51.76%	44.89%	44.15%	48.68%
UNSAT	Longer	2,030	147	172	206	222	261	268	387	333	34	0
	Shorter	5,249	928	822	751	732	677	656	457	205	21	0
	Total	7,279	1,075	994	957	954	938	924	844	538	55	0
	Speedup Possibility	72.11%	86.33%	82.70%	78.47%	76.73%	72.17%	71.00%	54.15%	38.10%	38.18%	100.00%
Time Out		29	0	3	0	0	0	1	3	12	8	2
Accuracy		45.36%	11.60%	18.01%	21.30%	21.55%	22.86%	23.93%	30.35%	54.77%	94.82%	99.84%

Table 15: Performance of CVC4 Solving Simplified PC

CVC4												
Ratio of Concretization	Total	50%	45%	40%	35%	30%	25%	20%	15%	10%	5%	0%
SAT	Longer	3,104	53	93	111	119	124	131	184	369	661	628
	Shorter	2,964	88	126	148	143	154	160	185	297	492	586
	Total	6,068	141	219	259	262	278	291	369	666	1,153	1,216
	Speedup Possibility	48.85%	62.41%	57.53%	57.14%	54.58%	55.40%	54.98%	50.14%	44.59%	42.67%	48.11%
UNSAT	Longer	2,058	158	178	211	227	259	273	385	334	33	0
	Shorter	5,221	917	816	746	727	679	651	459	204	22	0
	Total	7,279	1,075	994	957	954	938	924	844	538	55	0
	Speedup Possibility	71.73%	85.30%	82.09%	77.95%	76.21%	72.39%	70.45%	54.38%	37.92%	40.00%	100.00%
Time Out		29	0	3	0	0	0	1	3	12	8	2
Accuracy		45.36%	11.60%	18.01%	21.30%	21.55%	22.86%	23.93%	30.35%	54.77%	94.82%	99.84%

We have two very important observations from the data. Logically, replacing symbolic variables with concrete values in a PC should have reduced the complexity and lead to a faster solution. However, to our surprise, we did not observe a consistent reduction of time cost after concretizing the PC. The chance for Z3 to get a speedup on concretized SAT PC is only 49.65%. While on UNSAT PCs the possibility is a little higher (72.11%), but there are still some cases that we took longer to solve the concretized PC. CVC4 showed a similar performance with the speedup possibility being 48.45% on SAT PCs and 71.73% on UNSAT PCs. Meanwhile, there is a 55% of chance

for a PC to become unsatisfiable after concretization, and the accuracy drops significantly as we concretize more symbolic variables. As we concretize 10% of the symbolic variables, we have 95% of the chance for the satisfiability does not change, and the possibility greatly drops to only 54% if we increase the ratio to 15%.

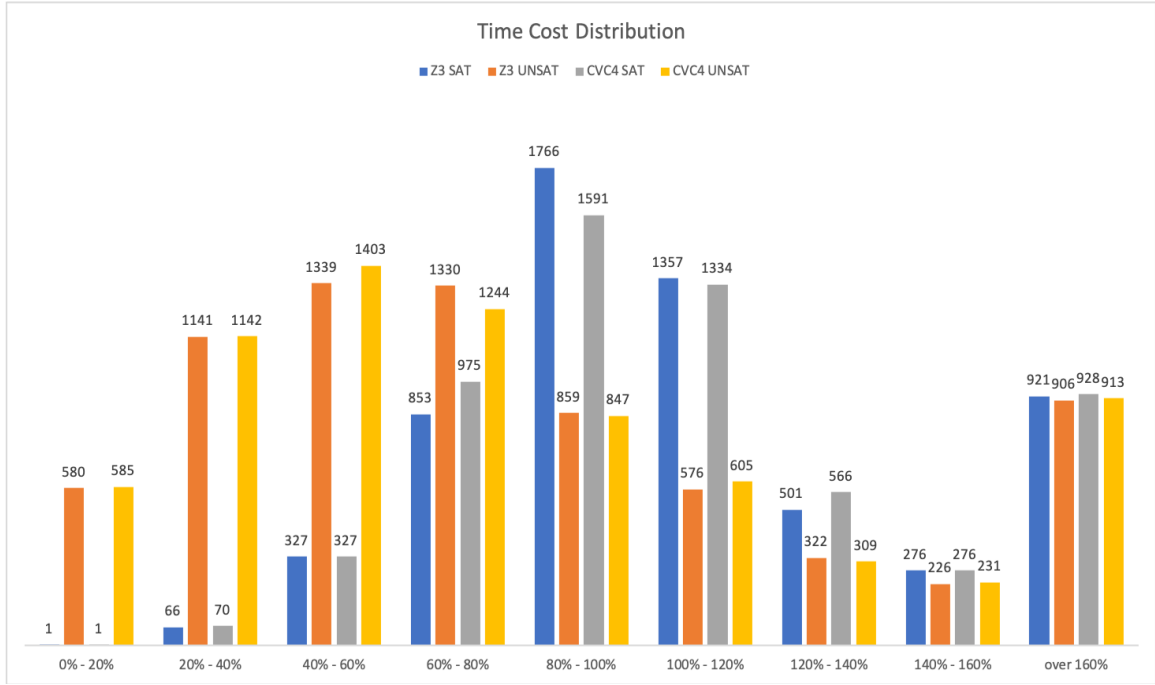


Figure 15: Time Cost Distribution

Clearly, these two observations make our technique non-applicable. At the first glance, it does not make sense for a concretized PC, which is simpler, needs more time to solve. Also, the low accuracy greatly challenges the soundness of the result from symbolic execution. Therefore, we need to know the reason behind these behaviors and find solution to these problems.

Solving Concretized Constraint in Parallel

Observed Problems

We identified two major problems which caused the unexpected results in the case study.

Random behavior of SMT solvers

The first question we need to answer is why the time cost could increase on the concretized PCs, which are supposed to be easier to solve. As we have a closer look at the evaluation results, we have noticed an even more surprising fact that the performance can be very different when solving exactly the same PC. Data in Table 14 and Table 15 to show an example of this behavior. As we set the ratio to 0%, no symbolic variable is actually replaced, thus the “simplified” PC stays identical with the original version. However, we noticed that there is a 50% chance to get a faster solving and vice versa. We found a similar behavior among all the PCs as we conducted more experiments. In the worst case, a PC that could be solved within one minute in one solving run may even end up passing the 2 hours’ time limit we set in another run without any change applied.

To understand this behavior, we looked into details of the solvers we choose, and find out it could be caused by the random behavior of SMT solving algorithms which is very common for modern SMT solvers including Z3 and CVC4. Based on the research paper “The Proof Complexity of SMT Solvers” (Robere et al., 2018), a typical SMT solver applies a collection of heuristics that interact with each other in very complicated ways, and it is difficult to discern which sets of heuristics are universally useful. As a result, off-the-shelf solvers usually embrace a non-deterministic strategy to choose the solving heuristic (or “model”). This is a reasonable solution for SMT solvers: since it is almost impossible to develop a fixed strategy to choose the heuristic for all constraints, introducing randomness can improve the possibility to solve a complex constraint in a shorter time. As for the two SMT solvers we used (Z3 and CVC4), even though the

documentation is limited, we find a related research (Reynolds et al., 2015) showing both solvers have the same randomness behavior.

We need to mention that this randomness exists for every application including when used in symbolic execution runs introduced in previous chapters. The reason why it is not observed before is that the PCs generated by a conventional program are relatively simple and are able to suppress the impact of choosing different solving heuristic. When a PC has a very small number of symbolic variables and constraints, normally it won't take a SMT solver a long time to find out a solution and there is not so much "space" for randomness in the first place. However, for PCs with a large number of constraints and symbolic variables, choosing different models can have a great impact on the performance. In our experiments, the solving time for one same concretized PC ranges from almost 0% of the original time cost to not able to solve it within 2 hours regardless of the final result. Meanwhile, this impact is not only on the time cost but also on the memory cost. We find that while some PCs can be solved given enough time (as we lift off the time limit of 2 hours), occasionally the solvers use up all 16 GB memory and result in a JVM error when solving the same PC. This shows that different heuristics chosen by SMT solvers impact on performance in terms of both time and space.

Change of satisfiability

The second problem that draws our attention is the change of satisfiability after replacing symbolic variables with a concrete value. In evaluation, we have a 55% of chance for the concretized PC to become unsatisfiable, and the rate increases as we replace more variables with concrete values in a PC. It is worth noting that it is not always wrong for the concretized PC to become unsatisfiable. On the contrary, it is a

reasonable behavior of concretization. With fewer variables kept symbolic, the solution space becomes smaller. As a result, it is possible to have an original satisfiable PC become unsatisfiable after replacing some variables, while an unsatisfiable will always remain unsatisfiable after concretization. Meanwhile, if a PC is unsatisfiable, there is a higher chance to get a faster solving result than the satisfiable PCs. To be more particular, the possibility for faster solving on satisfiable PCs is approximately 49%, while on unsatisfiable PCs the possibility is around 71%. We anticipate the reason as by using concrete values, it is possible to introduce logic short circuits into the PC e.g., some constraints can be identified as false in a very short time.

Although the change of satisfiability being expected, in our research, we would rather employ a more conservative thinking on the applicability of our technique, for being a symbolic execution approach, we need to make the result of constraint solving sound by all means. That is to say, we need to find a way to address this low accuracy problem. Since we cannot foresee whether an incoming PC is satisfiable or not before concretization, in order to maintain the soundness of the result, all unsatisfiable results must be double-checked, similar to DeepSolver.

Parallelization on Constraint Solving

As we described, randomness of SMT algorithms is widely accepted and adopted by solvers. In addition, our technique is implemented in a well-developed symbolic execution framework which uses solvers as black-boxes and changing the solver itself may introduce unexpected side effects. Thus, currently we don't aim a complete removal of such non-deterministic behaviors. Instead, we would like to employ parallelization to minimize the impact of the randomness. To be more specific, we assign multiple threads

(noted as “Cocoa threads”) to solve a copy of the same concretized PC simultaneously to increase the possibility of getting a fast solving heuristic. If any of these threads returned a solution, all other Cocoa threads can be stopped. In this way, the risk for choosing a heuristic that ends in a long solving time is much lower. For instance, let us say we assign 10 threads to solve a simplified PC. If the chance for choosing a “bad” solving strategy for this PC is 50% (which is a conservative high number for an assumption), the possibility for all threads ending in this category is only 0.098%.

Meanwhile, parallelism can also be used to address the problem of low accuracy. In addition to major threads, we also launch several “backup threads” that are set to solve the original PC. If a major thread returns the concretized PC to be satisfiable, we consider it a reliable result and stop all threads. Otherwise, if a Cocoa thread returns unsatisfiable as result, we stop all the running major threads which are solving the same concretized PC and wait for a report from a backup thread. No matter the backup thread reports the PC to be satisfiable or not, it can be trusted for these threads are solving the original PC. Note that in order to minimize the impact of randomness of SMT solvers, we need to run more than 1 backup threads so that the time cost won’t change too much from not using our technique at all.

Evaluation

We name our solution with parallelization as Cocoa, which represents concretization for constraint analysis. Since the major purpose of Cocoa is to optimize constraint solving, we ask the following two research questions for evaluation:

- **RQ1:** How efficient Cocoa comparing with solving PC without concretization?
- **RQ2:** How does the number of concretized symbolic variables impact the

performance of Cocoa?

Experiment Setup

We introduce additional evaluation runs to examine the performance of the parallel solution. We randomly collected 75 PCs from multiple DNN programs (MNIST2, MNIST16, CIFAR10). All PCs have different complexity for each of these DNNs have different structure and takes different number of inputs. We still use ABS-VAL as our heuristic, and for each PC we apply four ratios to decide the number of symbolic variables for concretization (5%, 10%, 15% and 20%).

For SMT solvers, we only use Z3 for this evaluation since we did not observe a clear difference in performance between Z3 and CVC4 in last evaluation.

We run each concretized PC with two different settings for comparison, both using 10 threads in total for solving. For experimental group (which we use Cocoa to solve constraints), we set 8 Cocoa threads which solve the concretized PC and 2 backup threads which solves the original PC, while for the comparison group all 10 threads are setup to solve original PCs.

Results and Analysis

Table 16, Table 17 and Table 18 show the time cost of solving a PC with Cocoa on different configurations, which is the time cost of getting the first reliable result (SAT from a Cocoa thread or the result from backup thread). We also report the time cost of the comparison group and calculate the speedup with Cocoa. For each subject, we report not only the time cost and speedup for each PC, but also the average time cost and speedup among all 25 PCs with a certain ratio of concretization. To have a clearer look of the

data, we use a box chart (Figure 16) to show the distribution of speedup of Cocoa comparing with the comparison group.

Table 16: Time Cost of Solving PCs from MNIST2

PC	Comparison Group	Experimental Group (MNIST2)							
		5%		10%		15%		20%	
	Time Cost (ms)	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup
pc-01	562,844	523,444	93.00%	503,029	89.37%	486,932	86.51%	560,029	99.50%
pc-02	151,050	140,477	93.00%	135,700	89.84%	151,805	100.50%	151,503	100.30%
pc-03	253,450	243,312	96.00%	232,119	91.58%	254,463	100.40%	252,436	99.60%
pc-04	253,450	235,708	93.00%	224,629	88.63%	215,868	85.17%	206,801	81.59%
pc-05	196,672	188,805	96.00%	179,364	91.20%	195,885	99.60%	197,458	100.40%
pc-06	299,243	281,288	94.00%	273,411	91.37%	263,841	88.17%	252,231	84.29%
pc-07	147,772	139,332	94.29%	132,922	89.95%	127,738	86.44%	120,712	81.69%
pc-08	266,389	250,405	94.00%	243,393	91.37%	236,091	88.63%	265,057	99.50%
pc-09	332,030	308,787	93.00%	296,435	89.28%	287,245	86.51%	332,362	100.10%
pc-10	42,490	40,791	96.00%	39,036	91.87%	37,318	87.83%	35,750	84.14%
pc-11	8,830	8,565	97.00%	8,785	99.49%	8,874	100.50%	8,794	99.59%
pc-12	199,958	187,961	94.00%	181,382	90.71%	175,940	87.99%	167,670	83.85%
pc-13	21,121	19,642	93.00%	19,249	91.14%	21,015	99.50%	21,184	100.30%
pc-14	138,663	128,956	93.00%	124,313	89.65%	139,356	100.50%	138,385	99.80%
pc-15	188,279	176,982	94.00%	170,256	90.43%	164,978	87.62%	155,574	82.63%
pc-16	251,690	237,794	94.48%	228,757	90.89%	222,809	88.53%	252,193	100.20%
pc-17	30,561	28,727	94.00%	30,622	100.20%	30,713	100.50%	30,438	99.60%
pc-18	233,407	217,068	93.00%	212,727	91.14%	203,792	87.31%	232,706	99.70%
pc-19	276,998	260,378	94.00%	249,963	90.24%	239,964	86.63%	276,444	99.80%
pc-20	289,141	271,792	94.00%	258,745	89.49%	252,276	87.25%	242,941	84.02%
pc-21	13,313	12,781	96.00%	12,193	91.59%	13,286	99.80%	13,273	99.70%
pc-22	172,707	160,617	93.00%	155,316	89.93%	148,792	86.15%	139,715	80.90%
pc-23	341,250	317,362	93.00%	301,811	88.44%	340,226	99.70%	340,567	99.80%
pc-24	20,921	19,874	95.00%	19,238	91.96%	20,816	99.50%	20,941	100.10%
pc-25	214,174	205,607	96.00%	197,999	92.45%	191,861	89.58%	213,531	99.70%
Average	196,256	184,258	93.89%	177,256	90.32%	177,275	90.33%	185,148	94.34%
Medium	199,958	188,805	94.00%	181,382	90.89%	191,861	88.53%	197,458	99.60%

- *RQ1: How efficient Cocoa comparing with solving PC without concretization?*

Our first observation from the bar chart is that there is no clear sign of out-liners.

Since the impact of randomness is minimized by parallelization on both Cocoa and the comparison group, even though we did not get as large reduction in time cost as shown in previous evaluation, the performance is more stable in general. We can observe a trend of performance that the more symbolic variables we concretized, the larger of reduction of constraint solving time can be expected. Although the overall speedup for

each configuration differs from subject to subject, we can get the largest speedup when replacing 10% to 15% of symbolic variables with concrete values.

Table 17: Time Cost of Solving PCs from MNIST16

PC	Comparison Group	Experimental Group (MNIST16)							
		5%		10%		15%		20%	
		Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup
pc-01	717,627	644,591	89.82%	670,224	93.39%	723,810	100.86%	723,320	100.79%
pc-02	171,593	145,131	84.58%	140,777	82.04%	204,527	119.19%	204,099	118.94%
pc-03	257,506	257,715	100.08%	258,474	100.38%	267,067	103.71%	284,432	110.46%
pc-04	325,937	286,212	87.81%	279,915	85.88%	276,276	84.76%	272,408	83.58%
pc-05	248,201	214,021	86.23%	210,169	84.68%	271,187	109.26%	286,323	115.36%
pc-06	337,247	318,102	94.32%	313,967	93.10%	308,630	91.51%	302,766	89.78%
pc-07	168,165	146,525	87.13%	144,474	85.91%	140,140	83.33%	137,197	81.58%
pc-08	330,856	281,718	85.15%	273,266	82.59%	265,068	80.12%	386,812	116.91%
pc-09	379,843	350,381	92.24%	340,220	89.57%	334,436	88.05%	442,579	116.52%
pc-10	41,641	34,998	84.05%	34,648	83.21%	34,232	82.21%	33,547	80.56%
pc-11	9,263	8,475	91.49%	9,824	106.06%	9,933	107.23%	10,987	118.61%
pc-12	219,554	203,125	92.52%	200,891	91.50%	195,266	88.94%	192,728	87.78%
pc-13	22,959	20,449	89.07%	19,999	87.11%	23,940	104.27%	24,455	106.52%
pc-14	137,832	123,400	89.53%	120,932	87.74%	159,160	115.47%	142,920	103.69%
pc-15	229,136	190,560	83.16%	185,987	81.17%	180,593	78.81%	176,981	77.24%
pc-16	327,197	276,531	84.52%	268,512	82.06%	260,725	79.68%	358,012	109.42%
pc-17	36,215	30,438	84.05%	30,103	83.12%	37,159	102.61%	38,847	107.27%
pc-18	260,949	220,467	84.49%	216,058	82.80%	210,440	80.64%	303,873	116.45%
pc-19	267,581	248,689	92.94%	243,218	90.90%	239,083	89.35%	235,019	87.83%
pc-20	348,994	311,827	89.35%	302,472	86.67%	293,398	84.07%	286,356	82.05%
pc-21	12,661	11,367	89.78%	11,219	88.61%	12,837	101.39%	14,112	111.46%
pc-22	214,157	183,945	85.89%	179,898	84.00%	176,300	82.32%	171,187	79.94%
pc-23	414,619	398,303	96.06%	389,142	93.86%	380,970	91.88%	414,479	99.97%
pc-24	20,294	17,777	87.60%	17,510	86.28%	17,072	84.12%	21,288	104.90%
pc-25	236,877	214,969	90.75%	209,165	88.30%	203,099	85.74%	250,801	105.88%
Average	229,476	205,589	89.59%	202,843	88.39%	209,014	91.08%	228,621	99.63%
Medium	236,877	214,021	89.07%	209,165	86.67%	204,527	88.94%	235,019	104.90%

Table 18: Time Cost of Solving PCs from CIFAR10

PC	Comparison Group	Experimental Group (CIFAR10)							
		5%		10%		15%		20%	
		Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup	Time Cost (ms)	Speedup
pc-01	1,788,155	1,662,982	93.00%	1,598,123	89.37%	1,546,983	86.51%	1,779,212	99.50%
pc-02	459,494	427,331	93.00%	412,799	89.84%	461,791	100.50%	460,872	100.30%
pc-03	933,040	877,056	94.00%	852,495	91.37%	822,656	88.17%	786,456	84.29%
pc-04	750,875	694,688	92.52%	687,047	91.50%	667,810	88.94%	659,130	87.78%
pc-05	1,142,899	965,923	84.52%	937,912	82.06%	910,712	79.68%	1,250,536	109.42%
pc-06	583,232	542,404	93.00%	524,502	89.93%	502,471	86.15%	471,818	80.90%
pc-07	839,995	709,683	84.49%	695,491	82.80%	677,406	80.64%	978,167	116.45%
pc-08	911,317	856,636	94.00%	832,647	91.37%	807,667	88.63%	906,760	99.50%
pc-09	1,140,855	1,060,992	93.00%	1,018,551	89.28%	986,974	86.51%	1,141,996	100.10%
pc-10	678,457	637,752	94.00%	615,429	90.71%	596,964	87.99%	568,904	83.85%
pc-11	421,674	392,155	93.00%	378,036	89.65%	423,782	100.50%	420,829	99.80%
pc-12	859,449	825,071	96.00%	787,116	91.58%	862,884	100.40%	856,010	99.60%
pc-13	753,760	700,996	93.00%	668,047	88.63%	641,991	85.17%	615,026	81.59%
pc-14	594,736	570,946	96.00%	542,397	91.20%	592,356	99.60%	597,113	100.40%
pc-15	769,416	726,936	94.48%	699,310	90.89%	681,127	88.53%	770,954	100.20%
pc-16	778,179	723,705	93.00%	709,232	91.14%	679,443	87.31%	775,842	99.70%
pc-17	940,576	884,139	94.00%	841,697	89.49%	820,654	87.25%	790,287	84.02%
pc-18	676,361	649,307	96.00%	625,281	92.45%	605,897	89.58%	674,331	99.70%
pc-19	580,001	505,365	87.13%	498,291	85.91%	483,343	83.33%	473,192	81.58%
pc-20	1,027,970	875,298	85.15%	849,037	82.59%	823,566	80.12%	1,201,825	116.91%
pc-21	799,532	743,083	92.94%	726,735	90.90%	714,380	89.35%	702,237	87.83%
pc-22	1,073,506	959,180	89.35%	930,404	86.67%	902,492	84.07%	880,831	82.05%
pc-23	723,851	621,734	85.89%	608,055	84.00%	595,894	82.32%	578,612	79.94%
pc-24	1,382,340	1,327,942	96.06%	1,297,399	93.86%	1,270,154	91.88%	1,381,873	99.97%
pc-25	63,216	55,375	87.60%	54,544	86.28%	53,179	84.12%	66,312	104.90%
Average	826,915	759,867	91.89%	735,623	88.96%	725,303	87.71%	791,565	95.73%
Medium	778,179	723,705	93.00%	699,310	89.84%	679,443	87.31%	770,954	99.60%

- *RQ2: How does the number of concretized symbolic variables impact the performance of Cocoa?*

Meanwhile, we can also see from the distribution that as the ratio set to 15%, occasionally Cocoa would take the similar or more time to solve a PC. Once the ratio is set to 20%, for all 3 DNNs the speedup bounced back to over 94% on average. This is due to the fact that as more symbolic variables increased, there is a higher chance for us to get the report from solving the original PC. In our case study, we have already discovered that the accuracy drops as the ratio increases, thus it is safe to anticipate that the overall speedup will continue dropping with a higher ratio of concretization. Among

all evaluation results, we can see that the worst performance happens on PC-02 of MNIST16 with 20% symbolic variables being concretized, which costs 118.94% time of the comparison group. We conjecture the reason being that since the result of Cocoa comes from a backup thread, and we only have 2 of them in setting, we did not get a better solving heuristic as among the 10 backup threads in comparison group. In practice, user may need to try different configurations to get the best speedup for different subjects. In the worst scenario, all PCs will be solved by a backup thread, and we can still expect the overall performance being similar to conventional constraint solving.

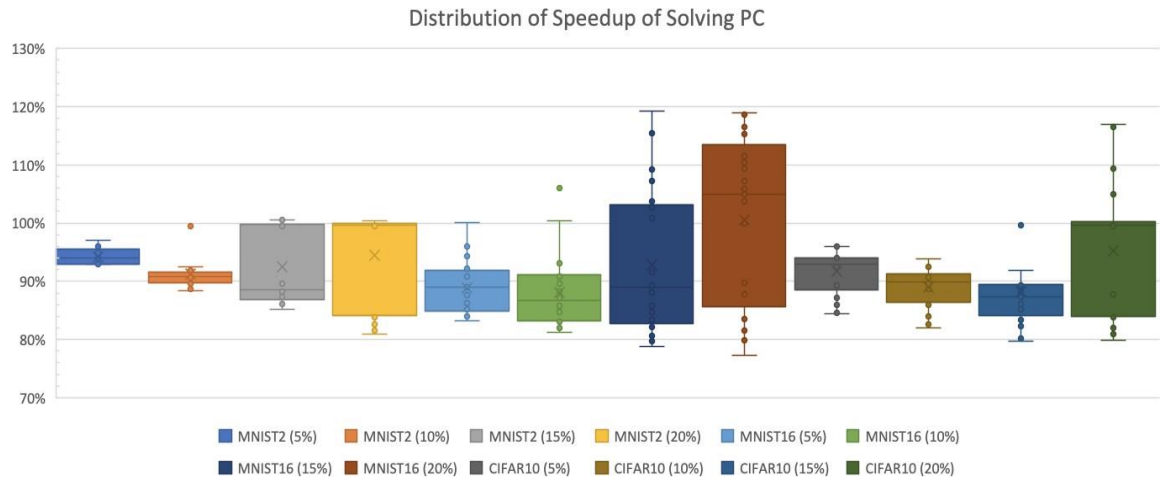


Figure 16: Distribution of Time Cost of Solving Concretized PC in Parallel

VII. RELATED WORK

Parallel Symbolic Execution and Guided Symbolic Execution

The area of parallel symbolic execution (Bucur et al., 2011; Staats and Păsăreanu, 2010; Kim et al., 2012a; Siddiqui and Khurshid, 2010, 2012; Qiu et al., 2018) has been widely studied. These techniques essentially divide the symbolic execution tree into sub-trees and distribute them to different workers to explore them in parallel. A primary difference between our work and previous work is no previous technique is specifically focused on property checking like our technique, which uses locations of assertions in the state space as “cut points” to distribute the exploration in order to get earlier reports about assertion violations. The parallel symbolic execution tool Cloud9 (Bucur et al., 2011) utilizes load balancing that first assigns the whole program to a worker. Whenever an idle worker becomes available, the load balancer instructs the busy worker to suspend exploration and breaks off some of its unexplored sub-tree to send to the idle worker to balance the workload. The workload is transferred among different workers as branches in the program.

SCORE framework (Kim et al., 2012a) distributes concolic testing (Sen et al., 2005) so that whole execution paths are generated one by one on distributed nodes in a systematic manner while preventing redundant test case. In concolic testing, a symbolic path formula is extracted on the path traversed during each concrete execution, and further symbolic path formulas are then generated by negating path condition. The concrete executions based on the values generated by solving these symbolic path formulas can traverse different new paths in the program. Rather than exploring these

paths sequentially in regular concolic testing, the SCORE framework employs distributed nodes to explore these paths in parallel.

Static partitioning (Staats and Păsăreanu, 2010) leverages an initial shallow symbolic execution run to minimize the communication overhead during parallel symbolic execution. It creates pre-conditions using conjunctions of clauses on path conditions encountered during the shallow run and restricts symbolic execution by each worker to explore only paths that satisfy the pre-condition.

Another parallel symbolic execution tool ParSym (Siddiqui and Khurshid, 2010) first executes the program on an initial input and collects the constraints along the executed path. By negating individual constraints in the path condition, new path conditions are generated and sent to distributed workers, so each worker explores different paths of the program in parallel. A symbolic execution monitor controls the work distribution and stops exploration when all path constraints are analyzed or a certain bound of constraint number is reached.

Several other projects (Siddiqui and Khurshid, 2012; Qiu et al., 2018) use ranges as the basis for parallel symbolic execution. Ranged symbolic execution (Siddiqui and Khurshid, 2012) introduced the idea of bounding a run of symbolic execution using a range defined by a pair of two ordered tests (t_1 , t_2) such that symbolic execution is restricted to program paths that are lexicographically between the path executed by t_1 and the path executed by t_2 . Rui et al. (Qiu et al., 2018) extended ranged symbolic execution with two new types of ranges, i.e., feasible ranges and unexplored ranges. These projects partition the workload using ranges with respect to an initial shallow depth exploration

and assign the ranges to available workers who perform a deeper exploration and use work stealing for load balancing.

Much work has been done for guiding symbolic execution (Santelices and Harrold, 2010; Yang et al., 2014b; Ma et al., 2011). Directed symbolic execution (Yang et al., 2014b) uses a def-use analysis to compute change affected locations and then uses this information to guide symbolic execution to explore only program paths that are affected by the changes. Santelices and Harrold (Santelices and Harrold, 2010) use control and data dependencies to symbolically execute groups of paths, rather than individual paths. Ma et al. (Ma et al., 2011) propose a call chain backward search heuristic to find a feasible path to the target location. Our work leverages reachability of properties to guide symbolic execution to only explore paths relevant to the checked properties.

Some recent projects (Guo et al., 2015; Yang et al., 2014a; Zhang et al., 2014) have explored more efficient checking of properties. Guo et al. (Guo et al., 2015) introduce assertion guided symbolic execution for eliminating redundant executions in multi-threaded programs to reduce the overall computational cost. An execution is considered redundant when it shares the same reason why it cannot reach the bad state with previous executions, and thus can be eliminated for the purpose of checking assertions. While it focuses on eliminating redundant executions for multi-threaded programs, our guided check focuses on eliminating irrelevant executions for single-threaded programs. iProperty (Yang et al., 2014a) computes differences between assertions of related programs in a manner that facilitates more efficient incremental checking of conformance of programs to properties. Our approach is orthogonal and can

use iProperty to compute differences between assertion versions when the checked assertion is changed, thus speeding up the assertion checking carried out by each worker. iDiscovery (Zhang et al., 2014) uses assertion separation to focus symbolic execution on checking one assertion at a time, and violation restriction to generate at most one violation of each assertion. While our work shares some insight with assertion separation on checking assertions separately, the guided and prioritized check in our work has potential to check each assertion more efficiently.

Machine Learning for Satisfiability Checking

Researches Arbelaez et al. (2010); Xu et al. (2011); Gent et al. (2010); Xu et al. (2011); Kadioglu et al. (2010); Pulina and Tacchella (2007); O’Mahony et al. (2008); Kotthoff (2014); Map (2020) have applied machine learning techniques to constraint satisfiability checking with different models and techniques including linear regression, decision tree learning, clustering, k-nearest neighbors, and so on. Deep learning-based solvers have recently been developed to improve the efficiency of satisfiability checking. Selsam et al. (2018) proposed NeuroSAT, a deep learning-based SAT solver using a single-bit supervision. It was developed to understand the extent to which neural networks can do precise logical reasoning. It classifies a SAT problem as satisfiable or unsatisfiable after a certain number of iterations when it finds a satisfying solution. To the best of our knowledge, most learning-based constraint solvers are designed to solve SAT problems, while symbolic execution frameworks usually apply SMT solvers for the path conditions that use a more general language model. DeepSolver currently supports linear integer arithmetic path conditions, which cannot be solved directly by a SAT solver.

Xu et. al Hong Xu and Kumar (2018) applied deep learning to predict the satisfiability of Boolean binary constraint satisfaction problems. Different from our approach, this approach uses randomly generated constraint satisfaction problems as training data and applied a convolutional neural network (CNN) as the deep learning model, while we take existing constraint solutions as training dataset and use a simpler DNN structure. Moreover, this approach aims to predict the satisfiability of Boolean binary constraints, while our approach classifies the satisfiability of linear integer arithmetic path conditions.

Another related work is Path Constraint Classifier (PCC) Wen et al. (2019). PCC is a deep learning model whose aim is to reduce overall constraint solving latency by dynamically selecting a solver per query. The goal is to be able to predict which solver is the best for a given path condition in terms of efficiency. DeepSolver on the other hand focuses on using deep learning to classify path conditions as satisfiable or not.

Reuse for Efficient Constraint Solving

Many techniques have been developed to speed up symbolic execution by reusing previous constraint solutions. For example, KLEE Cadar et al. (2008) optimizes constraint solving by an approach named counterexample caching. With the cached constraint solving results, KLEE can quickly check satisfiability of a path condition if it is a similar query to one of the stored records: if a path condition has a subset that is already known as unsatisfiable, it is unsatisfiable as well. Similarly, if a path condition has an already known satisfiable superset in the cache, it is satisfiable.

Green (Visser et al., 2012) applies Redis in-memory database to maintain the constraint solutions and uses slicing and canonizing to path conditions in order to

increase the reuse rate. To further improve Green, GreenTrie (Jia et al., 2015) stores constraints and solutions into L-Trie, which is indexed by an implication partial order graph of constraints and is able to carry out logical reduction and logical subset and superset querying for given constraints. Compared to Green and GreenTrie, our approach reuses the collective knowledge of constraint solutions: once the DNN is trained, DeepSolver does not reuse individual constraint solution, and can quickly classify the satisfiability of path conditions as long as they can be transformed into the required form of matrix. Similar to Green, PLATINUM (Zheng et al., 2020) reuses constraint solving results in checking Alloy (Jackson, 2012) specifications. Alloy specifications are usually translated by Kodkod (Torlak et al., 2013) into a Boolean formula, which is then solved by a SAT solver. Instead of solving it directly, PLATINUM slices and canonizes it into a normalized format, and searches whether an equivalent record exists in the storage. If such a slice exists, previous results will be reused.

Unlike techniques that store path conditions and their satisfiability information, memoized symbolic execution (Yang et al., 2012) stores positions and choices taken during symbolic execution in a trie – an efficient tree-based data structure. When applied to regression analysis, the trie guided symbolic execution would potentially skip exploration of portions of program paths, whereas symbolic execution using our approach would only skip calls to the underlying constraint solver. Our approach could work together with memoized symbolic execution to provide a fast classification of path conditions when program paths cannot be skipped by memoized symbolic execution. Some techniques, e.g. (Makhdoom et al., 2014), take advantage of test suites to reduce expensive constraint solving calls typically in regression testing. While these approaches

reuse existing test cases for efficient regression testing, our technique reuses constraint solving results to efficiently classify path conditions encountered in symbolic execution.

VIII. FUTURE WORK AND DISCUSSION

This chapter discusses several directions for future work.

Hybrid of STAPAR and STASE

Although the evaluation results have demonstrated the efficiency of STAPAR and STASE, there is still space for further improvement of efficiency of property checking using symbolic execution. Specifically, STASE uses dynamic analysis to partition the workload, and thus the worker in the first stage may explore paths and states that are irrelevant to property checking, and it may cause a delay in starting workers in the second stage. Meanwhile, for STAPAR, since each worker is assigned for checking a sub-version, currently we do not have a finer granularity for the workload balancing even though the exploration is optimized with guided and prioritized checking. We consider a hybrid of STAPAR and STASE leveraging the strength of each technique could address these shortcomings in both and can reach a better solution to distribute the workload for parallel property checking with symbolic execution.

Further Improvement of DeepSolver

While working on DeepSolver, we designed the DNNs with two structures with different layers and neurons. The evaluation result shows the two DNN structures does not have significant impact on the accuracy while the small DNN can classify path conditions faster. In future, we would like to explore more types of DNN structures, for example, with different connection between neurons and different activation functions, aiming to find a better DNN structure for higher accuracy as well as better performance. We would also like to explore the impact of using a single universal matrix for all PCs instead of grouping PCs based on the size of matrices. Furthermore, currently our

algorithm supports linear integer arithmetic path conditions. We would like to expand the support for path conditions involving other theories.

More Evaluation for Cocoa

For Cocoa, we generated 75 path conditions from three DNNs to evaluate the performance of the technique. In practice, the number of paths in a DNN is usually tremendous, which means 25 PCs may not be enough to expose the best configuration in the heuristics for a DNN. Also, we currently do not have enough DNN programs in Java at hand for evaluation, as Cocoa is currently implemented on top of Symbolic Pathfinder. In future, we plan to further optimize our algorithm of Cocoa, and search for both DNN programs and conventional programs to further evaluate the performance of Cocoa.

Testing Deep Neural Networks

Due to the limitation of SMT solvers, the technique Cocoa introduced in Chapter VI is less stable and efficient than we expected. However, as we mentioned earlier in that chapter, the ranking of symbolic variables can be used to improve symbolic execution in other applications. For instance, we can use the importance score to guide a fuzz testing by mutating the most important pixels in an image. If the mutated input is classified a different result than the original one, we consider the input an interesting input (or “a faulty case”). Technically, these guided fuzzing for DNNs can be expected to detect faulty cases faster than randomly fuzzing the same input. We have implemented the methodology and are currently conducting experiments to evaluate it. During our experiments, we noticed that unlike conventional programs, generating test cases for deep neural networks can be far more challenging. There are two major problems:

The first problem is how to decide whether a test suite is sufficient or not. Usually a test suite can be evaluated based on different coverage criteria (Hemmati, 2015) such as statement coverage, branch coverage, path coverage, etc. This evaluation is based on an assumption that all statements/branches/paths in a program are purposely created by the programmer. For instance, if a programmer writes an if-statement in the program, both branches are assumed to be executed with some inputs. Thus, generating tests to cover both branches can be helpful: if a test case can be generated, programmer can use it to execute the program and see whether the software behaves as he expected; if such a test case cannot be generated, programmer can get a report about the infeasible branch and check whether there is a problem in the code. In the recent years, researchers have proposed various coverage criteria such as path coverage (Wang et al., 2019) and neuron coverage (Xie et al., 2019) to evaluate test generation techniques for DNNs. However, these coverage criteria, despite being applicable, are not as persuasive on DNN testing. The key reason is that unlike regular programs a DNN is not manually programmed based on a clear specification. In contrary, the overall goal of deep learning is to automatically extract features whose structure is “unknown.” As a result, the problem of being non-interpretable remains a big problem in deep learning techniques (Gilpin et al., 2018). Therefore, we have little idea about the “meaning” of each neuron and path in a DNN, and thus we cannot decide whether a coverage criterion is suitable or not. In other words, if a certain path in a DNN model is missed in testing, we do not know whether it means the test suite is not complete or that path is in fact “not meant to be covered.”

That said, we can still generate test cases to cover all paths in a DNN with symbolic execution regardless of the meaning of each path. However, another problem

arises as not all test inputs generated by symbolic execution are meaningful for a DNN. Take a DNN trained on MNIST database as an example. A valid input for the DNN should contain the same features of the training data. In some cases, the generated data can be decided invalid based manually — for example, if an image is not a white digit with black background, one can easily tell that the data are invalid. However, there are also cases that can be arbitrary, especially if the input comes from fuzz testing. Consider an image generated by lowering the brightness of an original image in MNIST without changing anything else. We end up with a question on how to reason about the classification result of the new image: with an image that is clearly “darker” than the training data, both classification results would make sense. If the classification result is correct, that is because the change did not affect the important features in the original image. If the classification result is wrong, it also makes sense for the DNN as it may not be trained to identify a “dark” image from the very beginning. As a result, one cannot avoid manual efforts to double-check the test cases.

Due to the aforementioned problems, automated detection of invalid/unreasonable inputs for DNNs is highly demanded for test generation techniques such as fuzzing and symbolic execution and is an interesting direction for future work.

Balancing the Training Dataset

Class imbalance problem is a common problem in deep learning area. In reality, the sizes of data in each class/category can be very different, for some data are statistically rare in the first place or are difficult to produce. For instance, the number of patients who have a rare disease is, with no doubt, much smaller than the number of people who do not. To address data balancing problem, two kinds of techniques are

widely developed: techniques to generate a balanced training dataset from real data, and techniques to train a DNN with limited and imbalanced data (Batista et al., 2004; D  ez-Pastor et al., 2015; Batista et al., 2003). For instance, one can apply different sampling techniques to discard some data in a larger category (under-sampling) or duplicate data in a smaller category (over-sampling). Some alternative ways include admitting the imbalance in dataset if it accurately represents the ratio of data in the real world or building a cost-sensitive classifier Hall et al. (2009) by setting the smaller category with less tolerance. In our work, we applied methods to generate new data based on existing data, which is similar to an over-sampling technique SMOTE (Bowyer et al., 2011).

However, as we took a deeper look into this class imbalance problem, we discovered that these widely accepted techniques have only considered “size balancing.” In other words, they only consider the size of each categories in the training dataset, assuming that if each category has the same amount of training data, no category would be ignored by the model as all categories are treated as equally important. However, deep learning is in fact a decision maker based on features, not the size of data, which was commented by Yoshua Bengio, one of the leading scientists in deep learning and a winner of Turing Award in 2018. In his paper (Bengio, 2012), he mentioned that “Deep learning algorithms seek to exploit the unknown structure in the input distribution in order to discover good representations, often at multiple levels, with higher-level learned features defined in terms of lower-level features.” Simply put, a deep learning model is in fact a feature extractor, and it extracts features from the input data and checks the relationship between the distribution of features with the final classification result. Thus,

simply balancing the training dataset based on the number of data records in each category can be insufficient.

Developing an approach to decide whether a dataset is balanced in terms of features can be very challenging. Yet, since deep learning is more and more widely considered a useful in software engineering, it is an interesting topic for future research.

IX. CONCLUSION

Symbolic execution is a powerful technique for checking properties such as assertions. However, it could be very expensive due to problems like path explosion and time-consuming constraint solving. In this dissertation, we introduce four of our works aiming to increase the scalability of symbolic execution by different approaches. Specifically, the first two works employ parallel programming to distribute workload to multiple processes, while the third and fourth work aim to address expensive constraint solving problem in symbolic execution.

The first part of this dissertation introduced a novel approach STAPAR for partitioning the problem of property checking using symbolic execution into simpler sub-checks where each check is focused on checking one single property. All sub-checks are performed by multiple workers in parallel for better scalability. The parallelized property checking enabled us to further optimize each sub-check by pruning irrelevant paths regarding the checked property. Moreover, check is prioritized to explore shorter paths towards properties so that earlier feedback on the checked property can be provided to the user. Experiments using five subject programs with assertions that are manually written as well as automatically synthesized, showed that STAPAR reduced the overall analysis time compared with regular non-parallel property checking; and in sub-checks which focus on checking one single assertion, our guided check pruned state space exploration and thus reduced the time cost, and our prioritized check provided earlier feedback compared to regular check.

The second part of this dissertation addressed property checking with symbolic execution in an alternative way. Normally, to check all assertions in a program, users are

usually forced to run symbolic execution sequentially for multiple times, or to wait for a long time before results of assertion checking are reported. In this work, we presented an approach STASE to checking assertions in two stages, so that the assertions are checked in parallel. The overhead due to redundancy of some state space exploration between workers is mitigated by memoization. We implemented STASE on top of Symbolic Pathfinder and conducted an evaluation with checking assertions in four different Java programs and demonstrated the effectiveness and efficiency of STASE. Overall, STASE detected the same assertion violations as sequential checking technique, while achieving up to 5.65X speedup. Moreover, due to the cost reduction as well as parallelism, the users get the assertion checking results much earlier in parallel checking than in sequential checking.

Our third part of this dissertation introduced DeepSolver, a novel approach to solve constraints based on deep learning, which leverages existing constraint solutions for training DNNs to classify path conditions for their satisfiability during symbolic execution. To the best of our knowledge, this is the first work that results in a fully functional and applicable solution to use deep learning on constraint solution reuse for symbolic execution. Our evaluation shows that DeepSolver is highly applicable with a high accuracy. It is more efficient than conventional constraint solving and multiple existing constraint solution reuse frameworks in classifying path conditions for satisfiability. Meanwhile, the evaluation results show that DeepSolver can well support overall symbolic execution task. For future work, we plan to further evaluate our approach on additional real-world artifacts and compare our solution with other constraint solution reuse techniques.

The last part of this dissertation in this dissertation aimed to optimize constraint solving with concretization, which means replacing less-important symbolic variables with concrete values so that the complexity of the constraint is reduced. We consider a typical scenario of running symbolic execution on a DNN for image classification as example so that we can develop multiple heuristics to calculate the importance score of symbolic variables. As our case study exposed a thread of the unstable performance caused by the randomness of solvers, we reinforced the solution with parallelization to reduce its impact as well as solving the low correct rate problem at the same time. The evaluation results show that our solution, Cocoa, can reduce the time cost of constraint solving with a carefully selected number of symbolic variables to concretize.

REFERENCES

- SIR Repository. <http://sir.unl.edu>.
- Lonestar cluster. <https://www.tacc.utexas.edu/systems/lonestar>.
- (2019). Choco. choco solver. <http://www.choco-solver.org/>.
- (2019). Redis. redis nosql database. <http://redis.io/>.
- (2019). The mnist database. <http://yann.lecun.com/exdb/mnist/>.
- (2019). Z3 theorem prover. <https://github.com/Z3Prover/z3/wiki/>.
- (2020). Cvc4. <https://cvc4.github.io/>.
- (2020). Deepsee data repository. <https://sites.google.com/view/deepsolver/>.
- (2020). Maplesat. <https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/maplesat>.
- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
- Acree, A., Budd, T., Demillo, R., Lipton, R., and Sayward, F. (1979). Mutation analysis. page 92.
- Agarap, A. F. (2018). Deep learning using rectified linear units (ReLU). *CoRR*, abs/1803.08375.
- Albert, E., Gómez-Zamalloa, M., Rojas, J. M., and Puebla, G. (2011). Compositional clp-based test data generation for imperative languages. In *LOPSTR 2011*.
- Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and Mcminn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001.
- Arbelaez, A., Hamadi, Y., and Sebag, M. (2010). Continuous search in constraint programming. In *ICTAI 2010*, pages 53–60.

- Avgerinos, T., Rebert, A., Cha, S. K., and Brumley, D. (2014). Enhancing Symbolic Execution with Veritesting. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1083–1094.
- Bagnara, R., Carlier, M., Gori, R., and Gotlieb, A. (2013). Symbolic path-oriented test data generation for floating-point programs. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pages 1–10.
- Barrett, C. and Tinelli, C. (2007). CVC3. In *CAV 2007*, pages 298–302.
- Batista, G., Prati, R., and Monard, M.-C. (2004). A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations*, 6:20–29.
- Batista, G. E. A. P. A., Bazzan, A. L. C., and Monard, M. C. (2003). Balancing training data for automated annotation of keywords: a case study. In *WOB*.
- Bengio, Y. (2012). Deep learning of representations for unsupervised and transfer learning. In Guyon, I., Dror, G., Lemaire, V., Taylor, G. W., and Silver, D. L., editors, *Unsupervised and Transfer Learning - Workshop held at ICML 2011, Bellevue, Washington, USA, July 2, 2011, volume 27 of JMLR Proceedings*, pages 17–36. JMLR.org.
- Borges, M., D’Amorim, M., Anand, S., Bushnell, D., and Păsăreanu, C. S. (2012). Symbolic execution with interval solving and meta-heuristic search. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, (1):111–120.
- Bowyer, K. W., Chawla, N. V., Hall, L. O., and Kegelmeyer, W. P. (2011). SMOTE: synthetic minority over-sampling technique. *CoRR*, abs/1106.1813.
- Bronkhorst, A. (2015). The cocktail-party problem revisited: early processing and selection of multi-talker speech. *Attention, perception & psychophysics*, 77.
- Bucur, S., Ureche, V., Zamfir, C., and Candea, G. (2011). Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198.
- Burnim, J., Juvekar, S., and Sen, K. (2009). WISE: Automated test generation for worst-case complexity. In *ICSE 2009*, pages 463–473.
- Cadar, C., Dunbar, D., and Engler, D. R. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*, pages 209–224.

- Cawley, G. C. (2012). Over-fitting in model selection and its avoidance. In Hollmén, J., Klawonn, F., and Tucker, A., editors, *IDA 2012*, pages 1–1.
- Chollet, F. et al. (2015). Keras. <https://github.com/fchollet/keras>.
- Christakis, M., Müller, P., and Wüstholtz, V. (2016). Guiding dynamic symbolic execution toward unverified program executions. *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 144–155.
- Clarke, L. A. (1976). A program testing system. In *ACM '76*, pages 488–491.
- Clarke, L. A. and Rosenblum, D. S. (2006). A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*.
- Corbett, J. C., Dwyer, M. B., Hatcliff, J., and Robby (2000). Bandera: a source-level interface for model checking java programs. In *ICSE*, pages 762–765.
- De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *TACAS 2008/ETAPS 2008*, pages 337–340.
- DÁez-Pastor, J. F., RodrÁguez, J. J., GarcÁa-Orsorio, C., and Kuncheva, L. I. (2015). Random balance: Ensembles of variable priors classifiers for imbalanced data. *Knowledge-Based Systems*, 85:96 – 111.
- Ernst, M. D. (2000). *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington.
- Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45.
- Finlayson, G. D., Schiele, B., and Crowley, J. L. (1998). Comprehensive colour image normalization. In Burkhardt, H. and Neumann, B., editors, *Computer Vision — ECCV'98*, pages 475–490, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.
- Gent, I. P., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N. C. A., Nightingale, P., and Petrie, K. (2010). Learning when to use lazy learning in constraint solving. In *ECAI 2010*, pages 873–878.
- Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., and Kagal, L. (2018). Explaining explanations: An approach to evaluating interpretability of machine learning. *CoRR*, abs/1806.00069.

- Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, volume 15 of Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR.
- Godefroid, P. (1997). Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97*, pages 174–186, New York, NY, USA. ACM.
- Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: directed automated random testing. In *PLDI 2005*, pages 213–223.
- Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks.
- Gopinath, D., Wang, K., Zhang, M., Păsăreanu, C. S., and Khurshid, S. (2018). Symbolic execution for deep neural networks. *CoRR*, abs/1807.10439.
- Guo, S., Kusano, M., Wang, C., Yang, Z., and Gupta, A. (2015). Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE*, pages 854–865.
- Hadarean, L., Bansal, K., Jovanovi, D., Barrett, C., and Tinelli, C. (2014). A tale of two solvers: Eager and lazy approaches to bit-vectors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8559 LNCS:680–695.
- Hall, M. A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18.
- Havelund, K. and Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.
- Hemmati, H. (2015). How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, pages 151–156. IEEE.
- Hong Xu, S. K. and Kumar, T. K. S. (2018). Towards effective deep learning for constraint satisfaction problems. In *CP 2018*, pages 588–597.
- Hossain, M., Do, H., and Eda, R. (2014). Regression testing for web applications using reusable constraint values. In *ICST 2014*, pages 312–321.

- Inkumsah, K. and Xie, T. (2008). Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE 2008*, pages 297–306.
- Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.
- Jaffar, J., Murali, V., and Navas, J. a. (2013). Boosting concolic testing via interpolation. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, pages 53–63.
- Jia, X., Ghezzi, C., and Ying, S. (2015). Enhancing reuse of constraint solutions to improve symbolic execution. In *ISSTA 2015*, pages 177–187.
- Johnson, J. and Khoshgoftaar, T. (2019). Survey on deep learning with class imbalance. *Journal of Big Data*, 6:27.
- Jones, J. A. (2008). *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, Atlanta, GA.
- Joshi, A. and Heimdahl, M. P. E. (2005). Model-based safety analysis of simulink models using scade design verifier. In *SAFECOMP 2005*, pages 122–135.
- Kadioglu, S., Malitsky, Y., Sellmann, M., and Tierney, K. (2010). ISAC–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756.
- Keogh, E. and Mueen, A. (2017). *Curse of Dimensionality*, pages 314–315. Springer US, Boston, MA.
- Kim, M., Kim, Y., and Rothermel, G. (2012a). A scalable distributed concolic testing approach: An empirical evaluation. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12*, pages 340–349, Washington, DC, USA. IEEE Computer Society.
- Kim, Y., Kim, M., Kim, Y. J., and Jang, Y. (2012b). Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. *Proceedings - International Conference on Software Engineering*, pages 1143–1152.
- King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.
- Kotthoff, L. (2014). Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60.
- Krawczyk, B. (2016). Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, 5(4):221–232.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90.

- Kuehne, H., Jhuang, H., Garrote, E., Poggio, T., and Serre, T. (2011). HMDB: A large video database for human motion recognition. In *ICCV 2011*, pages 2556–2563.
- Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. (2005). How the design of jml accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, pages 185–208.
- LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–44.
- Li, Y., Su, Z., Wang, L., and Li, X. (2013). Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices*, 48(10):19–32.
- Lloyd, J. and Sherman, E. (2015). Minimizing the Size of Path Conditions Using Convex Polyhedra Abstract Domain. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5.
- Ma, K.-K., Phang, K. Y., Foster, J. S., and Hicks, M. (2011). Directed symbolic execution. In *SAS*, pages 95–111.
- Makhdoom, S., Khan, M. A., and Siddiqui, J. H. (2014). Incremental symbolic execution for automated test suite maintenance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 271–276. ACM Press.
- Meyer, B., Nerson, J.-M., and Matsuo, M. (1987). Eiffel: Object-oriented design for software engineering. In *ESEC*, pages 221–229.
- Mohamed, A., Dahl, G. E., and Hinton, G. (2012). Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):14–22.
- Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *ICML 2010*, pages 807–814.
- Ngo, M. N. and Tan, H. B. K. (2007). Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07*, pages 215–224, New York, NY, USA. ACM.
- Nielsen, M. A. (2018). Neural networks and deep learning.
- Noller, Y., Kersten, R., and Păsăreanu, C. S. (2018). Badger: Complexity analysis with fuzzing and symbolic execution. *CoRR*, abs/1806.03283.
- Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378.
- O’Mahony, E., Hebrard, E., Holland, A., Nugent, C., and O’Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving. In *AICS 2008*.

- Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehltz, P., and Rungta, N. (2013). Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 391–425.
- Păsăreanu, C. S. and Rungta, N. (2010). Symbolic PathFinder: Symbolic execution of java bytecode. In *ASE 2010*, pages 179–180.
- Pulina, L. and Tacchella, A. (2007). A multi-engine solver for quantified boolean formulas. In Bessière, C., editor, *CP 2007*, pages 574–589.
- Qiu, R. (2016). *Scaling and certifying symbolic execution*. PhD dissertation, University of Texas at Austin.
- Qiu, R., Khurshid, S., Păsăreanu, C. S., Wen, J., and Yang, G. (2018). Using test ranges to improve symbolic execution. In Dutle, A., Muñoz, C., and Narkawicz, A., editors, *NASA Formal Methods, NFM 2018*, pages 416–434, Cham.
- Qiu, R., Khurshid, S., Păsăreanu, C. S., and Yang, G. (2017). A synergistic approach for distributed symbolic execution using test ranges. In *ICSE '17 - Companion*, pages 130–132.
- Qiu, R., Yang, G., Păsăreanu, C. S., and Khurshid, S. (2015). Compositional symbolic execution with memoized replay. In *ICSE 2015*, pages 632–642.
- Reynolds, A., Blanchette, J. C., Cruanes, S., and Tinelli, C. (2015). Model finding for recursive functions in smt. In *IJCAR*.
- Robere, R., Kolokolova, A., and Ganesh, V. (2018). The proof complexity of smt solvers. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification*, pages 275–293, Cham. Springer International Publishing.
- Rojas, J. M. and Păsăreanu, C. S. (2013). Compositional symbolic execution through program specialization. In *BYTECODE 2013 (ETAPS)*.
- Romano, A. and Engler, D. (2013). Expression reduction from programs in a symbolic binary executor. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7976 LNCS:301–319.
- SAE-ARP4761 (1996). *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International.
- Santelices, R. and Harrold, M. J. (2010). Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA*, pages 195–206.
- Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2018). Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685.

- Sen, K. and Agha, G. (2006). CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV 2006*, pages 419–423.
- Sen, K., Marinov, D., and Agha, G. (2005). Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA. ACM.
- Seo, H. and Kim, S. (2014). How We Get There: A Context-guided Search Strategy in Concolic Testing. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 413–424.
- Shrikumar, A., Greenside, P., Shcherbina, A., and Kundaje, A. (2016). Not just a black box: Learning important features through propagating activation differences. *CoRR*, abs/1605.01713.
- Siddiqui, J. H. and Khurshid, S. (2010). ParSym: Parallel symbolic execution. In *ICSE*, pages V1–405 – V1–409.
- Siddiqui, J. H. and Khurshid, S. (2012). Scaling symbolic execution using ranged analysis. In *OOPSLA*, pages 523–536.
- Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. (2008). Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2).
- Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP 2013*, pages 1631–1642.
- Soomro, K., Zamir, A. R., and Shah, M. (2012). UCF101: A dataset of 101 human actions classes from videos in the wild. *CoRR*, abs/1212.0402.
- Souza, M., Borges, M., d’Amorim, M., and Păsăreanu, C. S. (2011). CORAL: solving complex constraints for Symbolic PathFinder. In *NFM*, pages 359–374.
- Staats, M. and Păsăreanu, C. (2010). Parallel symbolic execution for structural test generation. In *ISSTA ’10*, pages 183–194.
- Tillmann, N. and De Halleux, J. (2008). Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP’08, pages 134–153, Berlin, Heidelberg. Springer-Verlag.
- Tolstikhin, I. O., Bousquet, O., Gelly, S., and Schölkopf, B. (2018). Wasserstein auto-encoders. *CoRR*, abs/1711.01558.
- Torlak, E., Taghdiri, M., Dennis, G., and Near, J. (2013). Applications and extensions of alloy: Past, present, and future. *Mathematical Structures in Computer Science*, 23:915–933.

- Visser, W., Geldenhuys, J., and Dwyer, M. B. (2012). Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA. ACM Press.
- Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. (2003). Model checking programs. *Automated Software Engg.*, pages 203–232.
- Wang, D., Wang, Z., Fang, C., Chen, Y., and Chen, Z. (2019). Deeppath: Path-driven testing criteria for deep neural networks. In *IEEE International Conference On Artificial Intelligence Testing, AITest 2019, Newark, CA, USA, April 4-9, 2019*, pages 119–120. IEEE.
- Wen, J. and Yang, G. (2018). Parallel property checking with symbolic execution. pages 554–603.
- Wen, J. and Yang, G. (2019). Parallel property checking with staged symbolic execution. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19*, pages 1802–1809, New York, NY, USA. ACM.
- Wen, S.-H., Mow, W.-L., Chen, W.-N., Wang, C.-Y., and Hsiao, H.-C. (2019). Enhancing symbolic execution by machine learning based solver selection.
- Whalen, M. W., Godefroid, P., Mariani, L., Polini, A., Tillmann, N., and Visser, W. (2010). FITE: future integrated testing environment. In *FoSER 2010*, pages 401–406.
- Willard, D. E. (1984). New trie data structures which support very fast search operations. *J. Comput. Syst. Sci.*, 28(3):379–394.
- Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., and See, S. (2019). Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In Zhang, D. and Møller, A., editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 146–157. ACM.
- Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Satzilla: Portfolio-based algorithm selection for SAT. *CoRR*, abs/1111.2249.
- Yang, G., Filieri, A., Borges, M., Clun, D., and Wen, J. (2019). Chapter five - advances in symbolic execution. volume 113 of *Advances in Computers*, pages 225 – 287. Elsevier.
- Yang, G., Khurshid, S., Person, S., and Rungta, N. (2014a). Property differencing for incremental checking. In *ICSE*, pages 1059–1070.
- Yang, G., Person, S., Rungta, N., and Khurshid, S. (2014b). Directed incremental symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 24(1):3:1–3:42.

- Yang, G., Păsăreanu, C. S., and Khurshid, S. (2012). Memoized symbolic execution. In *ISSTA 2012*, pages 144–154.
- Yi, Q., Yang, Z., Guo, S., Wang, C., Liu, J., and Zhao, C. (2015). Postconditioned symbolic execution. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*.
- Zhang, L., Yang, G., Rungta, N., Person, S., and Khurshid, S. (2014). Feedback-driven dynamic invariant discovery. In *ISSTA*, pages 362–372.
- Zhang, X.-S. (2000). *Introduction to Artificial Neural Network*, pages 83–93. Springer US, Boston, MA.
- Zheng, G., Bagheri, H., Rothermel, G., and Wang, J. (2020). Platinum: Reusing constraint solutions in bounded analysis of relational logic. In Wehrheim, H. and Cabot, J., editors, *Fundamental Approaches to Software Engineering*, pages 29–52, Cham. Springer International Publishing.