Meta-Solving via Machine Learning for Automated Reasoning

by

Joseph Scott

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Martin Müller Professor, University of Alberta
Supervisor:	Vijay Ganesh Adjunct Professor, University of Waterloo Full Professor, Georgia Institute of Technology
	Joanne Atlee Full Professor, University of Waterloo
Internal Members:	Mei Nagappan Associate Professor, University of Waterloo
	Nancy Day Associate Professor, University of Waterloo
Internal-External Member:	Derek Rayside
	Associate Professor, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation is based on first-authored peer-reviewed publications and preprints. Specifically, Chapters 3, 4, 5, and 6 all correspond to a first-author publication.

These publications appeared in conference and journal proceedings published by Springer. Springer's policy on reuse of published materials in a dissertation is as follows:

> Authors have the right to reuse their article's Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published article in their thesis according to current citation standards.

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Chapter 3

1. Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. "MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 303–325. DOI: 10.1007/978-3-030-72013-1_16

2. Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. "Algorithm selection for SMT". in: *Int. J. Softw. Tools Technol. Transf.* 25.2 (2023), pp. 219–239. DOI: 10.1007/s10009-023-00696-0. URL: https://doi.org/10.1007/s10009-023-00696-0

Chapter 4

1. Joseph Scott, Guanting Pan, Elias B. Khalil, and Vijay Ganesh. "Goose: A Meta-Solver for Deep Neural Network Verification". In: *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International*

Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022. Ed. by David Déharbe and Antti E. J. Hyvärinen. Vol. 3185. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 99–113. URL: https://ceur-ws.org/Vol-3185/extended678.pdf

Chapter 5

1. Joseph Scott, Federico Mora, and Vijay Ganesh. "BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers". In: *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Ed. by Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel. Vol. 12549. Lecture Notes in Computer Science. Springer, 2020, pp. 68–86. DOI: 10.1007/978-3-030-63618-0_5. URL: https://doi.org/10.1007/978-3-030-63618-0_5

2. Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. "BanditFuzz: Fuzzing SMT Solvers with Multi-agent Reinforcement Learning". In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 103–121. DOI: 10.1007/978-3-030-90870-6_6. URL: https://doi.org/10.1007/978-3-030-90870-6_6

Chapter 6

1. Joseph Scott, Guanting Pan, Elias B. Khali, and Vijay Ganesh. "Pierce: A Testing Infrastructure for Neural Network Verification Tools". In: Submitted to VSTTE 2023. 2023

Abstract

Automated reasoning (AR) and machine learning (ML) are two of the foundational pillars of artificial intelligence (AI) and yet have developed largely independently. The integration of these two sub-fields holds the tremendous potential to address problems that are otherwise difficult to solve, especially in the context of logic solvers, which are black-box deductive reasoning engines designed to tackle NP-Hard problems. The early 2000s witnessed a 'silent revolution' leading to the emergence of highly efficient boolean satisfiability (SAT), satisfiability modulo theories (SMT), and mixed-integer linear programming (MILP) solvers, capable of scaling to hundreds of millions of variables and being deployed billions of times daily in various industries. These advancements were primarily due to novel symbolic reasoning techniques as well as the use of ML in solvers. Building on previous successes, this thesis presents several advances in the use of ML in solvers.

A particular way of characterizing the value of using ML in the context of automated reasoning tools is the following: under widely believed complexity-theoretic assumptions, we do not expect any one solver or even a fixed sequence of solvers to perform well on all classes of instances. In fact, there is considerable empirical support for the aforementioned observation. Hence, it is reasonable for us to research methods that enable solver users to adaptively select a (sequence of) solver(s) for any given instance. ML provides a promising means to realize such (adaptive) algorithm selection methods.

We make the following contributions in this thesis: First, inspired by the success of the algorithm selection tool SATZilla for SAT solvers, we present the design and implementation of MachSMT, an algorithm selection tool for SMT solvers. MachSMT supports the entirety of the SMT-LIB and leverages ML over state-of-the-art SMT solvers. We provide empirical evidence for the value of algorithm selection and efficacy of MachSMT over three broad SMT usage scenarios, namely, solver selection for instances obtained from SMT-COMP (an annual competition for SMT solvers), configuration selection for a given solver (cvc5) over a large industrial benchmark suite, and finally for solver selection for a specific domain (network verification).

Second, we present the design and implementation of a novel adaptive algorithm selection tool (aka, a *meta-solver*), called `Goose`, for neural network verification solvers, a class of tools aimed at improving the trustworthiness of ML systems. Traditional algorithm selection tools (e.g., MachSMT) typically tend to be non-adaptive, i.e., once a solver is selected for a given instance this selection is not changed at runtime. By contrast, a key novelty here is that `Goose` implements an *adaptive* sequential portfolio, i.e., it calls a set of subsolvers in a sequence, wherein the order in which subsolvers are called is determined

adaptively based on information from their online and offline performance histories. We have implemented a variety of complete and incomplete subsolvers in `Goose` (in addition to using a set of off-the-shelf ones), and the following synergizing techniques to implement its adaptive sequential portfolio: algorithm selection, probabilistic satisfiability inference, and time-iterative deepening.

Additionally, in the spirit of improving solver performance via ML techniques, we present BanditFuzz, an RL algorithm for relative performance fuzzing of solvers. While MachSMT and `Goose` leverage supervised learning to make solvers faster, BanditFuzz leverages RL to search for performance issues in solvers. BanditFuzz searches for short problem instances for which a set of target solvers is under-performant, while a set of reference solvers is performant. Such instances expose performance issues in solvers, and are often caused by solver developer errors (e.g., missing rewrite rules, errors in heuristics, etc.). We additionally introduce `Pierce`, a versatile and extensible testing tool aimed at solvers for the neural network verification (NNV) problem. At its core, `Pierce` implements a fuzzing engine over the Open Neural Network Exchange (ONNX) – a standardized model format for deep learning and classical ML, and VNN-LIB – a specification standard over the input-output behavior of ML systems. `Pierce` supports the entirety of the VNN-LIB and most of ONNX v18.

Acknowledgements

In reflection, my time at the university has been transformative, largely due to my mentors and colleagues, of which I have many to thank.

First and foremost, the biggest mentor of my life, Vijay Ganesh, my first supervisor, who profoundly changed my life and world views, altered how I think at a deep and very fundamental level, and taught me how to conduct research.

Additionally, Joanne Atlee, my second supervisor, who graciously took me in late in the process, helped push me to the finish line and gave me confidence when I felt overwhelmed and drowned in self-doubt.

I would like to thank the rest of my internal committee, namely, Mei Nagappan, Derek Rayside, and Nancy Day, for their feedback throughout this entire process and their patience with me. Lastly, my external examiner, Martin Müller, for agreeing to be on my committee.

I would like to thank Aina Niemetz and Mathias Preiner for their invaluable mentorship. Aina and Mathias are the best engineers I have ever met. During my time in the program, I often found myself trying to emulate your work style and rigor. Additionally, for teaching me many of the soft skills of academia and how to deal with all the emotions and loneliness of graduate school.

Additionally, I would like to thank Federico Mora. I worked with Fed very early in my graduate studies while he was in Toronto. You really "woke" me up to what is required of being a grad student and the amount of unbelievable hardwork that is required for a very junior researcher to make contribution.

I would like to thank the many colleagues and collaborators I got to interact with here at Waterloo. Namely, Saeed, Piyush, Jimmy, Ed, Ian, Tony, Zack, Curtis, Reza, Murphy, Amin, Behkish, Elias, Jolly, John, Brian, Laura, Vineel, Dmitry, Hari, DJ, Sebastian, Maysum, Trishal, Hammad, Fa Fa, Rafeal, and Joy.

I would like to thank my colleagues from Ohio University. Namely, Sam, Charlie, Pat, Ana, Kellie, Mac, Evan, Jake, Daniel, Chad, Kevin, Taffie, Logan, Ryan, Kyle, Adam, and Elijah, and the local ACM chapter community at large. Additionally, my mentors from Ohio, namely Razvan Bunescu, David Juedes, David Chelberg, Frank Van Graas, Konstantinos Vasiliadis, Frank Drews, and Ralph Kelsey.

I would like to thank the Computer Science department at large for their support, specifically Nadine and Paula, for helping me with all the administrivia.

Finally, I would like to thank my mother, Betty, my sister, Marie, and my girlfriend, Cris, for being my backbone and giving me the strength to survive grad school.

Dedication

To Zachary (2000-2018).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI), at its core, is bifurcated into automated reasoning (AR) and machine learning (ML), two domains that, despite their foundational importance, have historically evolved independently. AR and symbolic AI, leverage logic solvers and search techniques for deductive reasoning, manifesting in advanced constraint solvers like Cadical [30], cvc5 [15], Gurobi [77], and tools such as WolframAlpha [191]. Concurrently, ML embodies the inductive reasoning aspect of AI, where patterns and models are inferred from data, utilizing algorithms such as neural networks and decision trees. This domain is powered by robust frameworks and libraries like PyTorch [127], scikit-learn [129], and XGBoost [47].

Logic solvers, with their roots in formal methods, exhibit remarkable strength in providing sound and robust solutions to well-defined problems (i.e., expressable in first-order logic). Their deductive power is capable of navigating through complex combinatorial search spaces to find satisfiable solutions or proofs/certificates of unsatisfiability. This makes them indispensable in domains where accuracy and certainty are critical, such as in verifying software correctness or solving intricate optimization problems. However, their strengths in deduction comes with a caveat – scalability. While logic solvers can sometimes scale to problems with hundreds of millions of variables on certain instances and benchmarks, these solvers often face significant challenges in scaling efficiently, which can lead to increased computational demands or even being infeasible for certain problem instances.

Incorporating ML techniques to enhance the performance of logic solvers, often by exploiting syntactic structure or runtime patterns, represents a pivotal shift towards overcoming inherent scalability challenges. ML, particularly through reinforcement learning (RL), has been instrumental in heuristically selecting and sequencing proof rules for im-

proved solver efficiency, as seen in the case of MapleSAT [101]. Additionally, ML aids in synthesizing strategies from tactics, exemplified by FastSMT [12], and in algorithm selection, as demonstrated by SATZilla [199]. The underlying thread connecting these diverse applications of ML within solvers is the recognition that it is unlikely that any single rule, algorithm, or strategy is universally effective across all problem instances.

This complexity-theoretic limitation necessitates an adaptive approach to solver selection, for which ML is a promising approach, as it is well established as the state-of-the-art for computational inductive reasoning. Moreover, the vast amounts of data generated by solvers (both offline information on large corpuses of instances and benchmarks and online information organically collected from the solving process) presents a rich resource for training useful and indicative ML models.

Thesis Statement. This thesis posits that one approach to overcoming the scalability challenges faced by logic solvers lies in innovative uses of machine learning (ML) to exploit syntactic and runtime patterns of solvers. Particularly, this thesis presents a case for meta-solving, which harnesses both online and offline information through machine-learning techniques. By integrating an adaptive approach to algorithm selection into the solving process, we demonstrate how the amalgamation of real-time performance data and historical patterns and interactions can significantly enhance solver efficiency and adaptability.

Contributions

- **MachSMT: Machine Learning-Driven Algorithm Selection for SMT solvers (Chapter 3)**. The rise of automated reasoning has led to the development of numerous efficient logic solvers for various theories and applications. The emergence of dozens of tools can be in part accredited to the empirical observation that there is no one solver or configuration that is best for a given instance or benchmark family. However, it can be intimidating for practitioners to determine which solver or configuration to use and when for a particular instance or benchmark. For example, consider the SMT theory of floating-point arithmetic comprised of logic solvers implementing a vast array of orthogonal algorithms, e.g., word-blasting [38], abstract CDCL [37], inter-reduction methods [146], and reduction to global optimization [63, 22]. Furthermore, consider the competition-winning variant of the cvc5 SMT solver[1].

[1] https://github.com/cvc5/cvc5/blob/smtcomp2021/contrib/competitions/smt-comp/run-script-smtcomp-current

On just a single logic (e.g., UF), 23 configurations of the cvc5 tool are used, in a statically ordered sequential portfolio, to determine the satisfiability of the given benchmark.

For a practitioner, it is a natural yet daunting task to determine which solver or configuration to select, or how to balance a portfolio of solvers that maximizes empirical efficiency for a given instance or benchmark. We present MachSMT, an algorithm selection tool for the entirety of SMT-LIB [19] and its standardized theories. Our tool uses machine learning (ML) to construct empirical hardness models (EHMs) and pair-wise comparators (PWCs) of solvers for algorithm selection and is designed to be easily tuned and extended by SMT solver users (see Chapter 3). We perform an empirical analysis of MachSMT across all divisions from the SMT-COMP 2019 and 2020. We observe that MachSMT improves on the best performing solvers in the competitions in 57 divisions, with up to 99.4% improvement in performance for the QF_BVFPLRA SQ'20 division, and up to 89.0% for the QF_UFBV SQ'19 division (see Section 3.3). We perform an empirical analysis of MachSMT to select from 23 different configurations of the cvc5 SMT solver [16] across multiple logics. We observe that MachSMT solves an additional 898 benchmarks over the competition version of cvc5 (Section 3.4). We perform an empirical analysis of MachSMT from three solvers (Bitwuzla [117], cvc5 [16], and Z3 [111]) to solve benchmarks from a network verification application. We evaluate MachSMT with and without domain-specific features (see Section 3.5). We observe MachSMT to considerably improve on the best standalone solver, and even more significantly with domain-specific features.

- **Goose: Meta-Solving for Deep Neural Network Verification (Chapter 4)** The success of tools like SATZilla and MachSMT at *offline* algorithm selection, i.e., leveraging offline collected data to select solvers or balance portfolios of solvers, has had great tremendous empirical success. However, it begs the question, is it possible to design an efficient adaptive ML-based sequential portfolio tool that dynamically changes its selections as it solves an instance by leveraging the underlying solvers' online and offline performance histories on any given instance?

  We developed `Goose`, a *meta-solver* for the verification of neural networks. By the term *meta-solver* we mean a tool that calls out a set of *subsolvers*, that get adaptively called in a sequence based on online and offline information collected about their performance histories on a given input, with the goal of discovering the sequence that optimally (e.g., least amount of time) solves the given input instance. This differs from the traditional non-adaptive algorithm (resp. sequential portfolio) selection, where the choice of a solver (resp. sequential portfolio) is selected, that

choice is not altered once the selection tool has made its choice. Goose implements three synergistic techniques. First is an ML-based *algorithm selection* technique that uses pre-trained empirical models for runtime prediction to construct an appropriate sequential portfolio of solvers and allocate resources among them. The empirical model leverages a feature vector that changes during the solving process from data collected by the other techniques. Second, we construct ML models for *probabilistic satisfiability inference*. More specifically, Goose leverages a recent result [160] that reduces a given neural network verification input to a disjunction over a set of subproblems. Given this syntactic structure, we use ML-driven empirical models to predict satisfiability and rank subproblems based on their likelihood of being satisfiable. The rich structure of neural network verification problems allows for the engineering of expressive subproblem features; this contrasts with more generic SAT/SMT benchmarks, where the originating domain may not be known. We show that attempting to solve the subproblems in an empirically-determined appropriate order can greatly influence solving times. Third and last, we use *time iterative deepening*, an exponentially increasing wall-clock timeout on a sequential portfolio. Two insights underlie this strategy: (a) trying out multiple solvers with short time limits first is a good time-saving strategy, assuming that there exists a solver in the portfolio that can quickly solve the subproblem; (b) the failure of all solvers within a given time limit provides valuable information on the difficulty of a subproblem, which then *adaptively* informs the next round of algorithm selection, under a larger time limit.

The key value addition of `Goose` is its meta-solver architecture (Figure 4.1), which implements an algorithm selection technique, a novel probabilistic satisfiability inference technique, and time iterative deepening that enables an adaptive sequential portfolio. See the algorithmic description in Section 4.3. We demonstrate the efficiency of `Goose` with a competition-like evaluation over more than 800 instances from VNN-COMP '21 against 13 competition solvers [40] including α, β-CROWN [183, 195, 207], Verinet [83], ERAN [163], and Marabou [91] (Section 4.4). We observe that `Goose` improves over the competition winner, α, β-CROWN, by 37.7% in `PAR-2` score (Figure 4.2, Table 4.3). We demonstrate the efficiency of `Goose` with a competition-like evaluation over more than 1500 instances from VNN-COMP '22 against 11 competition solvers (Section 4.4). We observe that `Goose` improves over the competition winner, α, β-CROWN, by 25.6% in `PAR-2` score (Figure 4.3, Table 4.4).

- **BanditFuzz: Fuzzing SMT Solvers with Multi-Agent Reinforcement Learning (Chapter 5)**. In MachSMT and `Goose`, we used machine learning with the aim of improving the efficiency of logic solving. Suppose instead one used machine learn-

ing to make logic solvers *slow* in interesting ways, as a aid in the development process? SMT solvers are prone to hard-to-find performance deficiencies. Although the worst-case complexity of problems solved by SMT solvers can be very high, they can be frustratingly slow on relatively simple formulas. Such performance deficiencies can be due to developer oversight (e.g., missing rewrite rules or unoptimized code and data structures) or the result of hard-to-entangle interactions of solver heuristics.

To this end, we built the BanditFuzz tool, a performance fuzzer for SMT solvers. The key insight to BanditFuzz is the formulation that the agent acts as the fuzzer while receiving a strong reward signal that comes organically over the course of a fuzzing loop. We evaluated BanditFuzz across 52 logics from SMT-COMP '20, targeting competition-winning solvers against runner-up solvers. BanditFuzz produced benchmarks exposing relative performance issues across SMT-LIB. We further provide several case studies demonstrating the utility of BanditFuzz to state-of-the-art SMT solver developers [157, 148]. Finally, several BanditFuzz results were reproduced independently, and the algorithm has been built into external fuzzers (e.g., SPRFinder [209]).

To the best of our knowledge, BanditFuzz is the first multi-agent RL performance fuzzing algorithm for SMT solvers that supports the entirety of SMT-LIB. The BanditFuzz tool includes two agents, one which learns how to mutate the best observed input [149] and another to help prevent the tool getting stuck in a local minimum (Section 5.2). We provide an implementation of the performance fuzzer BanditFuzz [149] and lift it to the entirety of the theories in the SMT-LIB initiative, namely, Arrays, Bit-Vectors, Booleans, Floating-Point, Integers, Reals, Strings, Uninterpreted Functions, and all combinations thereof in both quantified and quantifier-free logics (Sections 5.2, 5.5, 5.6). To test BanditFuzz, we perform an extensive empirical evaluation across all 52 logics that were tested in SMT-COMP '20, with the aim of finding benchmarks where competition winners are slow relative to runner up solvers. We provide to the community a set of 1500 benchmarks across all logics, exposing relative performance issues in state-of-the-art competition-winning SMT Solvers. To validate the efficiency of BanditFuzz, we baseline it against random fuzzing and observe that BanditFuzz consistently outperforms random fuzzing by up to a 82.6% increase in `PAR-2` margins (Section 5.2). To further demonstrate the usefulness of BanditFuzz, we include three case studies of BanditFuzz being used by solver developers and contributors. Specifically, developers were able to use BanditFuzz to find performance issues in the Z3 [111], CVC4 [21], and Bitwuzla [118, 119] SMT Solvers (Section 5.6).

- **Pierce: A Testing Tool for Neural Network Verification Solvers (Chapter 6)** We present `Pierce` tool for developing testing tools for Neural Network Verification (NNV) solvers (see Section 6.2 and architecture diagram in Figure 6.2). `Pierce` provides its users with two key capabilities. First, it has a computation graph generator that can generate graphs that correspond to random graphs, decision trees, and NNs (including those that are VNN-LIB compliant), whose shape (e.g., number of inputs and outputs, width, height, number of parameters, etc.) can be specified by the user using a configuration file. Second, `Pierce` has a mutator (i.e., a set of well-defined mutation operators) that mutates computation graphs in a well-formed manner. These and other capabilities make `Pierce` a highly configurable, extensible, and easy-to-use tool (Section 6.2.2) with many possible use cases (Section 6.2.3). For example, a user of `Pierce` can write a relatively straightforward script to configure `Pierce` into a debugging or performance fuzzer for NNV solvers.

  We configure `Pierce` into a performance fuzzer that is aimed at finding small instances that expose performance issues in a target solver relative to a set of reference solvers (Section 6.3). For example, we found instances that resulted in significant empirical slowdowns for four state-of-the-art neural network verification solvers, namely ERAN, Marabou, MIPVerify, and nnenum [9, 91, 163, 176]. This is done via a reinforcement learning fuzzing loop based on recent work [157]. The instances our fuzzer generated show a slowdown of 13.3x of the target solver (e.g., ERAN) relative to a set of reference solvers (e.g., Marabou, MIPVerify, and nnenum). We provide a rich suite of benchmarks of over 10,000 competition-grade instances, unit, and regression tests in ONNX format produced using `Pierce` (Section 6.2.3). The benchmarks are broad, covering a variety of problem classes and configurations.

Limitations. This thesis presents four distinct tools, each with its own set of limitations. Despite the unique challenges each tool faces, several common themes can be identified and abstracted to provide a holistic understanding of their limitations.

Firstly, data availability and quality significantly impact the performance of these tools. MachSMT, for instance, struggles with small sample sizes in certain logics, affecting its ability to perform well compared to standalone solvers. Similarly, `Goose` encounters issues when solving problems that differ significantly from its training data, and neural network verifiers like `Goose` struggle with very large inputs, failing to scale to the size of modern large language models (LLMs).

Secondly, efficiency and performance are recurrent concerns. BanditFuzz, for example, faces substantial time requirements to achieve results due to the nature of performance fuzzing and the complexity of modeling with sophisticated MDPs. Additionally, `Pierce`'s

tech stack is often brittle and actively being developed, leading to challenges in ensuring consistent performance.

Thirdly, the encoding of problem instances and the loss of information during this process is a common issue. Both MachSMT and `Goose` suffer from substantial information loss in their problem encodings, affecting their overall effectiveness. BanditFuzz also experiences issues with problem encoding, particularly in the context of SMT encoding.

Lastly, the adaptability and scalability of these tools remain a challenge. While `Goose` is trained on a fixed dataset, it may struggle with new problem instances, and `Pierce` is not suited for testing instances like LLMs. Moreover, BanditFuzz's benchmarks, while initially effective, may lose their relevance as solver developers fix discovered issues, affecting the permanence of its results.

Organization

The rest of this thesis is structured as follows, Chapter 2 overviews the preliminaries of this thesis, namely a survey of logic solvers, machine learning, and neural network verification. Chapter 3 presents MachSMT the ML-driven Algorithm selection tool for SMT solvers. Chapter 4 introduces `Goose`, a meta solver for deep neural network verification. Chapter 5 introduces BanditFuzz a reinforcement learning guided performance fuzzer for SMT Solvers. Chapter 6 presents `Pierce` a testing tool for neural network verification solvers. Finally, Chapter 7 concludes this thesis.

Chapter 2

Preliminaries

In this chapter, we go over important preliminaries on the satisfiability problem and relevant machine learning concepts.

2.1 Satisfiability

Satisfiability, often referred to as the heart of computational logic, is a fundamental concept in computer science. Its origins trace back to the classical decision problem in mathematical logic, evolving into a critical component of modern computational theory. At its core, satisfiability asks a seemingly simple yet profoundly complex question: given a formula, often written in some mathematical logic, does it have a solution, e.g., what assignment to the inputs cause it to evaluate to true?

The importance of satisfiability extends beyond theoretical interest. In practical terms, it is instrumental in a myriad of applications ranging from software verification and artificial intelligence to cryptographic systems and combinatorial optimization. The notion of finding a satisfying assignment, or proving its absence, forms the crux of many decision-making processes in computational tasks.

Satisfiability is a powerful tool for problem reduction in computational logic. It allows for the transformation of various computational problems into SAT problems, where the solution to the SAT problem is directly related to the original problem. This method is particularly effective in logic, enabling the simplification of complex structures and theorems into SAT formulations. A prime example of this is in implication checking or validity checking, where the objective is to determine if a set of premises Σ logically entails a

conclusion ψ. In SAT terms, this is achieved by expressing whether the conjunction of all premises in Σ and the negation of ψ results in a contradiction. If the resulting formula is unsatisfiable, it implies that Σ indeed entails ψ. This leads to the following SAT representation:

$$\Sigma \models \psi \iff \bigwedge_{\phi \in \Sigma} \phi \wedge \neg\psi \text{ is } \texttt{UNSAT}$$

This chapter lays the groundwork for understanding the nuances of satisfiability. It explores its theoretical underpinnings, delves into the complexities of solving satisfiability problems, and sets the stage for the more advanced discussions on Satisfiability Modulo Theories (SMT) and the integration of machine learning techniques in later chapters. As we progress, the chapter aims to illustrate not only the challenges inherent in satisfiability but also the innovative strategies and solutions that have been developed to tackle these challenges.

2.1.1 Boolean Satisfiability

Boolean Satisfiability, commonly known as SAT, is a fundamental problem in computational logic and computer science. It entails identifying a valid truth assignment for Boolean variables that satisfies a Boolean formula, typically constructed using logical operators such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). In essence, a SAT problem is structured as a series of clauses in conjunctive form, each clause being a disjunction of literals, with each literal being either a variable or its negation.

The designation of SAT as an NP-complete problem is of significant relevance. It indicates that, although verifying a solution to a SAT instance is relatively straightforward, finding that solution can be difficult. This distinction was first formally recognized by Stephen Cook [54] and Leonid Levin [178].

With the development of SAT solvers, considerable progress has been made in addressing the challenges presented by SAT problems. These solvers, which are either algorithmic or software-based, have been crafted to resolve SAT problems with greater efficiency. They utilize a variety of methods, including backtracking, heuristic approaches, and the Conflict-Driven Clause Learning (CDCL) technique. CDCL, in particular, has been instrumental in enhancing solver efficiency by learning from conflicts encountered during the problem-solving process and using this information to prevent similar issues in the future.

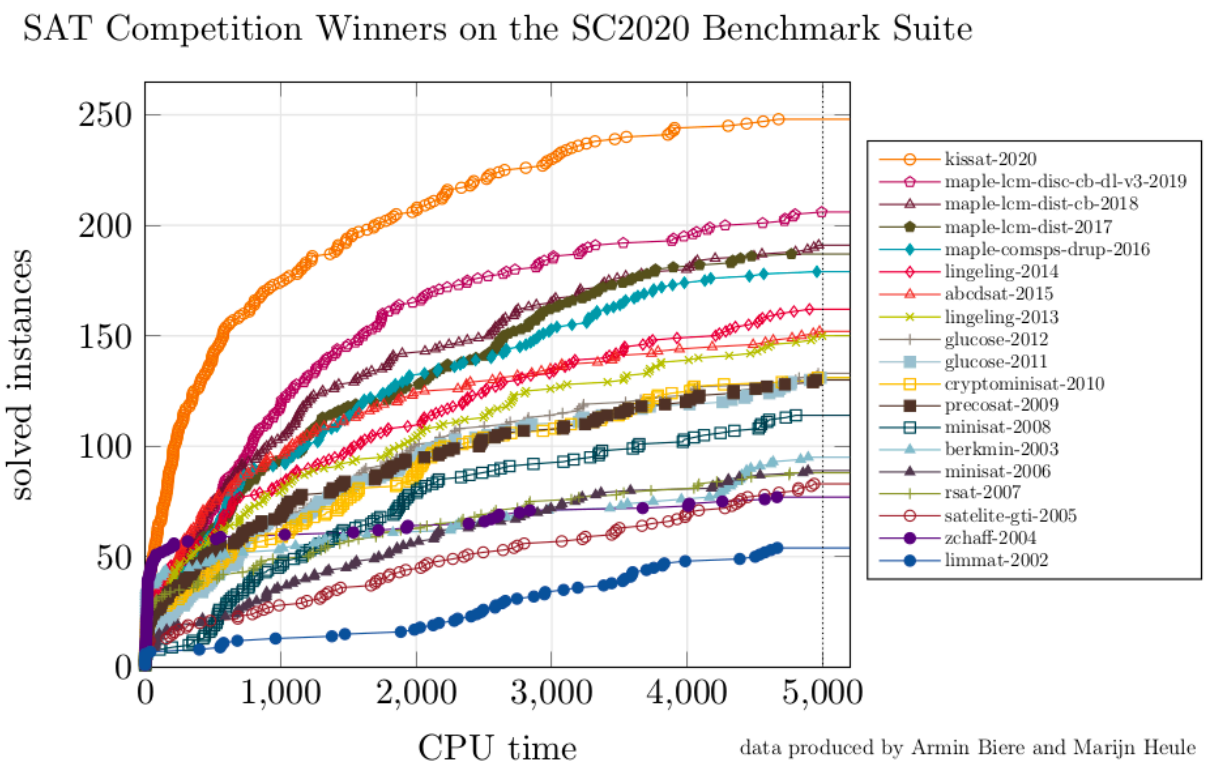


Figure 2.1: CDF Plot of SAT competition winning solvers from 2002-2020.

Contemporary logic solvers such as MiniSat [166], Glucose [7] MapleSAT [102], and [60], represent significant advancements in this field. Each of these solvers brings a unique set of optimizations and features to the table. MiniSat, with its minimalistic approach, has been influential in the field. Glucose, an offshoot of MiniSat, emphasizes intensive clause learning. Cadical stands out for its modern heuristics and streamlined design. MapleSAT integrates machine learning techniques to adapt its strategies dynamically, while Kissat is renowned for its efficiency and scalability.

These solvers exemplify the continuous evolution in SAT solving, highlighting the integration of advanced algorithmic strategies, heuristic methods, and adaptive learning techniques. Their ongoing development and refinement are critical in tackling the complexities of SAT problems, demonstrating the practical applications of these theoretical concepts in various domains. Figure 2.1 presents a CDF plot showing the progress of SAT solvers over the last 18 years.

2.1.2 Mixed Integer Linear Programming

Mixed Integer Linear Programming (MILP) represents a sophisticated extension of satisfiability problems, evolving from the foundations of Linear Programming (LP) and Integer Linear Programming (ILP). LP deals with optimizing a linear objective function subject to linear equality and inequality constraints, where the variables can take any continuous values. However, in many real-world scenarios, decision variables are discrete, necessitating the transition to ILP, where variables are constrained to integer values. MILP combines LP and ILP elements, allowing for continuous and integer variables in the same model. This mixed nature greatly enhances the expressive power and flexibility of MILP compared to traditional LP and ILP, making it more suitable for complex decision-making scenarios.

In contrast to Boolean Satisfiability (SAT) solving, which deals with finding assignments to Boolean variables that satisfy a given formula, MILP tackles a broader class of problems. While SAT solving is binary in nature, MILP accommodates a wider range of values and more complex relationships between variables. This difference in expressivity allows MILP to model and solve problems that are not just about logical consistency but also involve optimizing a certain objective, like minimizing costs or maximizing profits.

State-of-the-art MILP solvers, such as CPLEX [56], Gurobi [77], and SCIP [26], have made significant progress in efficiently solving large-scale and complex MILP problems. These tools employ advanced algorithms like branch-and-cut, which combines the branch-and-bound method with cutting planes to improve the solution process. Additionally, sophisticated heuristics are used for faster convergence to optimal or near-optimal solutions.

The applications of MILP are vast and diverse, extending far beyond the scope of traditional SAT problems. MILP is used in areas like resource allocation, production planning, supply chain management, and logistics, where decisions involve both quantitative optimization and adherence to logical or physical constraints. The expressive power of MILP, capable of modeling linear relationships and handling both continuous and discrete variables, makes it an indispensable tool in these domains.

2.1.3 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) is an extension of the Boolean Satisfiability (SAT) problem into the realm of first-order logic, making it a superset of SAT. While SAT focuses on finding assignments to Boolean variables within propositional logic constraints, SMT operates within a richer framework, dealing with more expressive first-order logic. This includes, but is not limited to, various mathematical theories like arithmetic, bit-vectors, arrays, and uninterpreted functions. In essence, SMT extends the Boolean logic of SAT with additional layers of theoretical constructs, enabling the modeling of more complex scenarios. In SMT, the addition of first-order logic elements similarly broadens the scope, allowing for the representation and solving of problems that are beyond the reach of traditional SAT, LP, and MILP methods.

At its core, SMT is concerned with determining the satisfiability of logical formulas within the context of specific mathematical theories. This advanced form of satisfiability combines traditional Boolean logic with elements of these theories, creating a multifaceted and expressive framework. This hybrid approach enables the accurate representation and analysis of problems involving detailed and nuanced constraints, which are commonplace in sophisticated computational tasks.

The integration of Boolean logic with other mathematical domains in SMT presents unique algorithmic challenges. Efficient SMT solvers rely on a combination of general SAT-solving techniques and theory-specific decision procedures. These theory-specific components, tailored to handle the intricacies of individual theories, are crucial for the effective processing of SMT instances. Furthermore, advanced techniques such as lazy theory combination and the use of efficient data structures are employed to manage the complexity and ensure the scalability of these solvers.

SMT has emerged as a powerful tool in areas like formal verification, automated theorem proving, and constraint solving, offering substantial improvements over traditional SAT in terms of expressiveness and applicability. The development and continual refinement of SMT solvers have been instrumental in integrating SMT into various software

development and verification frameworks, highlighting its growing relevance in both theoretical and practical aspects of computer science.

Efficient SMT solvers like Z3 [111] and cvc5 [16] have become cornerstones in the field, known for their robustness and versatility in handling a wide range of SMT problems. These tools have facilitated significant advancements in the automation of reasoning and verification processes, underscoring the critical role of SMT in advancing the frontiers of computational logic and satisfiability.

2.2 Machine Learning

2.2.1 Supervised Learning

Supervised learning in machine learning entails training a model on a labeled dataset, where the model learns to predict outputs (targets) from inputs (features). The objective is to approximate the mapping function so well that when the model encounters new, unseen data, it can accurately predict the output.

Mathematically, this can be represented as learning a function $f : X \to Y$, where X is the input space and Y is the output space. Given a training set $\{(x_1, y_1), ..., (x_n, y_n)\}$, where x_i represents the features and y_i the corresponding label for each instance, the goal is to learn a function $\hat{f}$ that approximates the true function f as closely as possible.

This process typically involves selecting a model (e.g., Ridge Regression, AdaBoost, Multi-Layered Perceptron, etc.), defining a cost function to evaluate the model (e.g., mean square error), and employing an optimization algorithm to adjust the model parameters to minimize the cost (e.g., Adam [92]).

2.2.2 Ridge Regression

Ridge Regression, an enhanced variant of linear regression, is engineered to address multicollinearity among the predictor variables by introducing a regularization factor, λ, into the traditional least squares estimation framework. This regularization term adds a penalty on the size of coefficients, thereby reducing model complexity and curtailing the risk of overfitting. The essence of Ridge Regression is encapsulated in the optimization problem it poses, aiming to minimize the penalized residual sum of squares:

$$\mathcal{L}(\beta) = \sum_{i=1}^{n}(y_i - X_i\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

where $\mathcal{L}(\beta)$ denotes the loss function, y_i the observed values, X_i the predictor matrix, β the vector of coefficients, n the number of observations, p the number of predictors, and λ the regularization parameter. By judiciously selecting the value of λ, Ridge Regression ensures a balanced trade-off between fitting the training data and maintaining a parsimonious model structure, thereby enhancing the model's generalization capability to unseen data.

AdaBoost (Adaptive Boosting)

AdaBoost, short for Adaptive Boosting, stands as a quintessential ensemble learning technique that synergistically amalgamates multiple weak learners, predominantly decision trees, to formulate a composite model of superior predictive prowess. The algorithm operates on a sequential learning paradigm, where it iteratively modifies the distribution of weights assigned to each observation in the training dataset. This adjustment is predicated on the accuracy of the preceding iteration, with increased emphasis placed on instances that were erroneously classified, thus progressively enhancing the model's accuracy on intricate datasets. Mathematically, the AdaBoost algorithm seeks to minimize the exponential loss function:

$$\mathcal{L}(\beta) = \sum_{i=1}^{n} \exp(-y_i \sum_{j=1}^{T} \alpha_j h_j(X_i))$$

where n denotes the number of training instances, T the number of iterations or weak learners, y_i the actual outcome, X_i the feature vector of the i-th instance, h_j the j-th weak learner, and α_j the weight attributed to the j-th weak learner, calculated based on its performance. Through this iterative refinement, AdaBoost effectively curtails both bias and variance, culminating in a robust model adept at tackling complex predictive tasks.

Multilayer Perceptron (MLP)

The Multilayer Perceptron (MLP), a cornerstone architecture within the domain of artificial neural networks, embodies a structured assemblage of nodes arrayed across multiple

layers, constituting an intricate directed graph wherein each successive layer is fully interconnected with its antecedent. Characterized by its capacity to effectuate non-linear data transformations through its layered architecture, the MLP is adept at unraveling complex patterns inherent in data, rendering it eminently suitable for a wide spectrum of tasks including, but not limited to, pattern recognition and classification.

Central to the MLP's learning mechanism is the backpropagation algorithm, a sophisticated method that facilitates the optimization of weights through iterative minimization of a predefined loss function, typically via gradient descent. This process involves propagating errors backward from the output layer to the input layers, thereby enabling the network to adjust its weights in a manner that incrementally reduces the discrepancy between the actual output and the desired output.

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

where Δw_{ij} denotes the adjustment to be made to the weight connecting the i-th node of one layer to the j-th node of the subsequent layer, η represents the learning rate, and $\frac{\partial \mathcal{L}}{\partial w_{ij}}$ is the partial derivative of the loss function $\mathcal{L}$ with respect to the weight w_{ij}.

Curse of Dimensionality.

The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces. As the dimensionality of the feature space increases, the volume of the space increases exponentially, making the available data sparse. This sparsity is problematic for any method that requires statistical significance, as it becomes difficult to sample the space sufficiently. It affects various aspects of model training and evaluation, leading to issues such as overfitting and increased computational complexity.

Cross Validation.

Cross-validation is a statistical method used to estimate the skill of a machine learning model on unseen data. It involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (the training set), and validating the analysis on the other subset (the test set). Common methods include k-fold cross-validation, where the dataset is divided into k subsets and the holdout method is repeated k times. This technique is essential for assessing the generalizability of predictive models.

2.2.3 Reinforcement Learning

There is a large literature on reinforcement learning and we refer the reader to the book by Sutton et al., on this topic [172]. In RL, an agent navigates an environment by taking actions to maximize the received reward. The multi-armed bandit (MAB) problem is a well-known RL problem based on a Markov Decision Process with a single state and a finite set of actions A. Since there is only a single state, MABs can be solved using computationally cheap algorithms relative to algorithms for other RL problems [172]. The agent solving the MAB computes an approximation of the probability distribution of rewards R over A. In the context of MAB, actions are often referred to as arms (or bandits). The term 'bandit' comes from gambling, as the arm or lever of a slot machine is referred to as a one-armed bandit, and MABs refer to several slot machines. The goal of the MAB agent is to maximize its reward by playing a sequence of actions (e.g., selecting which band/lever to pull).

In practice, MAB problems are commonly modelled so rewards are sampled from an unknown Bernoulli distribution (e.g., rewards are in $\{0, 1\}$). The MAB agent attempts to approximate the expected value of reward from the Bernoulli distribution for each action in A. Over time, the agent uses these distributions to form a *policy* – a stochastic process of how to select actions from A. This policy must remain privy to the exploration/exploitation trade-off, i.e., an MAB algorithm selects every action an infinite number of times, but selects the action(s) with the highest expected reward more frequently.

There are several algorithms for the MAB problem. In this chapter, we exclusively consider *Thompson Sampling*. In Thompson Sampling, an agent maintains a Beta distribution for each action in the action space A. Beta distributions are derived from Gamma distributions, and have a long history with numerous applications. We refer the reader to Gupta et al. on Beta and Gamma distributions [75]. In the context of Thompson Sampling, a Beta distribution acts as continuous model of the expected value of a Bernoulli distribution. It is maintained by two shape parameters α the samples of 1, and β the samples of 0, from the underlying Bernoulli distribution. The agent selects an action by sampling each action's Beta distribution and greedily picks the action based on the maximum over the sampled values. Upon taking the action, α is incremented on reward, otherwise β is incremented. For more on Thompson sampling we refer to Russo et al. [143].

2.2.4 Unsupervised and Semi-Supervised Learning

Unsupervised learning, a paradigm of machine learning, engages with data devoid of labeled responses, aiming to discern underlying patterns or structures from such datasets.

Contrarily, semi-supervised learning operates on a hybrid dataset comprising both labeled and unlabeled data, leveraging the intrinsic structure of the data to improve learning efficiency and prediction accuracy, especially when labeled data are scarce or expensive to obtain.

Within the domain of unsupervised learning, Principal Component Analysis (PCA) emerges as a pivotal linear dimensionality reduction technique. PCA seeks to identify the orthogonal axes (principal components) that maximize the variance in the dataset, thereby reducing its dimensionality while retaining as much of the variability in the data as possible. The mathematical foundation of PCA involves the eigen decomposition of the data covariance matrix or the singular value decomposition (SVD) of the data matrix itself, leading to:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{X}$ is the data matrix, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices representing the left and right singular vectors, and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values, which are directly related to the variance captured by each principal component. The principal components thus derived serve as a transformed coordinate system that optimally describes the variance in the data, facilitating data visualization, noise reduction, and the efficient preprocessing for other machine learning tasks.

Chapter 3

Machine Learning based Algorithm Selection for SMT Solvers

This chapter presents MachSMT, an algorithm selection tool for Satisfiability Modulo Theories (SMT) solvers. MachSMT supports the entirety of the SMT-LIB language and standardized SMT-LIB theories, and is easy to extend with support for new theories. MachSMT deploys machine learning (ML) methods to construct both empirical hardness models (EHMs) and pairwise ranking comparators (PWCs) over state-of-the-art SMT solvers. Given an input formula in SMT-LIB format, MachSMT leverages these learnt models to output a ranking of solvers based on predicted runtimes.

We provide an extensive empirical evaluation of MachSMT to demonstrate the efficiency and efficacy of MachSMT over three broad usage scenarios on theories and theory combinations of practical relevance (e.g., bit-vectors, (non-)linear integer and real arithmetic, arrays, and floating-point arithmetic). First, we deploy MachSMT on state-of-the-art solvers in SMT-COMP 2019 and 2020. We observe MachSMT frequently improves on the best-performing solvers in the competition, winning 57 divisions outright, with up to a 99.4% improvement in `PAR-2` score and median improvement of 9.8%. Second, we evaluate MachSMT to select configurations from a single underlying solver. We observe that MachSMT solves 898 more benchmarks and up to a 93.4% improvement in `PAR-2` score across 23 configurations of the SMT solver cvc5. Last, we evaluate MachSMT on domain-specific problems, namely network verification with simple domain-specific features, and observe an improvement of 77.3% in `PAR-2` score.

The artifact for this chapter is available at https://zenodo.org/record/7383299.

3.1 Motivation

Satisfiability Modulo Theories (SMT) solvers are tools to decide the satisfiability of formulas over first-order theories, such as the theories of bit-vectors, floating-point arithmetic, integers, reals, strings, arrays, and their combinations. Prominent examples of SMT solvers are Bitwuzla [117], Boolector [120], CVC4 [21], cvc5 [16], MathSAT [53], SMTInterpol [52], STP [65],Yices [59], and Z3 [111, 21]. In recent years, SMT solvers have had a revolutionary impact on applications in software engineering (broadly construed), such as software testing [44, 125] and verification [64, 39, 71, 99], as well as in sub-fields of AI [140, 89, 73]. This impact is a driver for an insatiable demand for evermore efficient solvers, not only to scale to larger instances obtained from existing applications (e.g., automatic bug-finding in commercial software [68, 8]), but also to solve problems from new application domains (e.g., verification and synthesis of cryptographic primitives [29]).

Motivation for Algorithm Selection for SMT Solvers. In response to this high demand, the SMT community has developed a plethora of solver heuristics and configurations. For example, in the 2019 edition [78] of the annual SMT-COMP competition [184], more than 50 solvers and their configurations were submitted. Many of these solvers implement very different algorithms to tackle the satisfiability problem for (a combination of) first-order theories, with significantly varying performance profiles. For example, for the quantifier-free theory of floating-point arithmetic (QF_FP), there exist several substantially different decision procedures, e.g., word-blasting [38], abstract CDCL [37], interreduction methods [146], and reduction to global optimization [63, 22]. In this specific setting of floating-point arithmetic, input instances may be derived from a variety of applications, such as software verification or analysis of machine learning (ML) models [155]. A very natural question arises in such a scenario: which solver or configuration is best for a given input instance?

Another well-known issue with many SMT solvers (even state-of-the-art ones) is that users may not know a priori which formula features or encoding would make an instance easy to solve. This can be very frustrating for users as they have to try a large number of different encodings with different solvers in order to find a combination that works best for their specific scenario, which may result in a combinatorial explosion. Users have also noted that as their applications change, what was once a great solver configuration in an earlier setting is suddenly underperforming in the newer one. One possible approach to address this problem is to use a portfolio of solvers, which has been successful in the context of SAT solvers. Unfortunately, given the plethora of SMT solvers and solver configurations (cvc5 [16] alone utilizes 23 different configurations in a sequential portfolio setting for quantified logics), such an approach quickly becomes infeasible.

One way to address the above-mentioned problems is to use an automated algorithm selection tool that can automatically and accurately predict the best algorithm from a given set of algorithms for a specific input. Such a tool selects the best SMT solver from a set of solvers for a given SMT formula. Note that in the following, in the context of algorithm selection, we will use *algorithm*, *solver configuration* and *solver* interchangeably when understood from the context.

Brief Overview of MachSMT. In this chapter, we introduce MachSMT, a machine learning-based algorithm selection tool for SMT solvers. MachSMT supports the entirety of the SMT-LIB language (v2.6) [20] and standardized SMT-LIB theories [19], and is easy to extend with support for new theories. MachSMT is written in Python and takes as input an instance for a specified logic (a combination of theories with or without quantifiers) of interest and outputs a ranking of solvers predicted to have the lowest runtime. Internally, MachSMT is a set of machine learnt models constructed by analyzing the runtimes of solver configurations on benchmarks with respect to a predefined set of features, as given in 3.1. These features include the frequency of occurrences of grammatical constructs (e.g., predicates, operators, bindings) and syntactical features that can have an influence on performance (e.g., quantifier nesting levels).

At its core, MachSMT utilizes two techniques for algorithm selection: empirical hardness models (EHMs) and pairwise ranking comparators (PWCs). In addition, it pipelines the set of features defined in Table 3.1 with feature preprocessing and Principal Component Analysis (PCA). By default, MachSMT uses scikit-learn [129] with Adaboosting for classification and regression models. It is, however, ML framework-agnostic with a highly abstract interface and a built-in PyTorch interface [127].

An EHM for a given solver is a mapping from an input instance to a predicted runtime of the solver on that input. At runtime, MachSMT queries all EHMs for all solvers that were considered during training on the given input and outputs a ranking of solvers based on their predicted runtimes (the top-ranked solver is predicted to solve the input problem the fastest). In contrast, a learnt PWC is a mapping that takes as input a pair of solvers and an input instance, and outputs a ranking over the input solvers based on which is predicted to have a lower runtime on the given input. During evaluation, MachSMT uses the learnt PWC as a comparator to rank the set of solvers on a given input.

While algorithm selection has been considered in the broad setting of satisfiability solvers (e.g., QBF solvers [133] and SAT solvers [198]) as well as certain specific SMT theories [156, 13, 186], we are not aware of previous work on algorithm selection aimed at the entirety of SMT-LIB [19] and its standardized theories. Our results demonstrate that the MachSMT algorithm selector is highly effective, as it outperforms the best performing

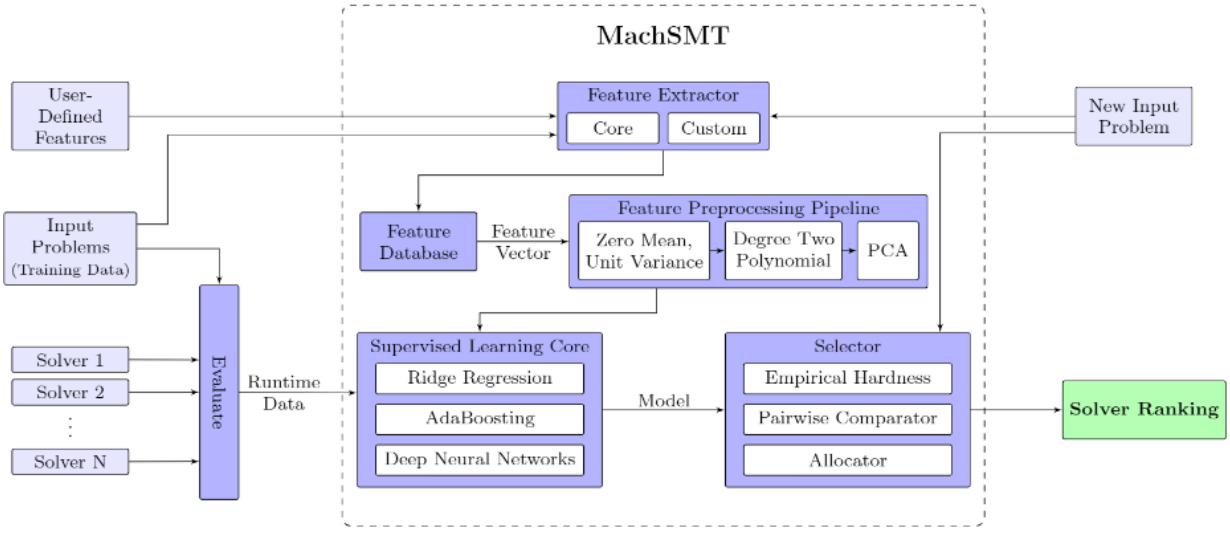


Figure 3.1: Architecture of MachSMT.

solvers in the competition on the majority of tracks from the SMT-COMP in 2019 and 2020.

Perhaps the first algorithm selection tool in the context of logic solvers was SATZilla [198]. Since its introduction, SATZilla has had a tremendous impact on SAT solver research, winning multiple gold medals in previous SAT competitions. MachSMT differs in several significant aspects from SATZilla. In particular, SATZilla deploys a feature selection scheme to avoid the curse of dimensionality, while MachSMT leverages a learnt dimensionality reduction scheme, Principal Component Analysis (PCA). In fact, a feature selection scheme would not scale in the context of SMT solvers given the very large number of learnt models that are incorporated into MachSMT.

It goes without saying that MachSMT is only as powerful as the underlying solvers that it has access to. MachSMT is clearly not a replacement for any particular SMT solver, but rather a tool that enables users to leverage the collective strength of the diverse set of algorithms and configurations implemented as part of these sophisticated solvers.

3.2 MachSMT

In this section, we provide an overview of the MachSMT tool. The architecture diagram of MachSMT is presented in Figure 3.1.

3.2.1 Features and Preprocessing

By default, MachSMT defines a feature vector with 196 entries (i.e., dimensions). This feature vector can optionally be extended with user-defined features. A complete description of each predefined feature is provided in Table 3.1. We use two strategies to mitigate taxing feature calculation times, which can severely impair algorithm selection solutions. First, all features are entirely syntactical properties of the input. This is a major difference between MachSMT and other algorithm selection solutions, such as SATZilla. Second, all features are calculated within a strict and user-adjustable time limit (default 10s). On a timeout, the feature value is recorded as -1.0.

By default, MachSMT performs three key preprocessing steps before constructing any learnt models over a given data set. The entirety of the ML pipeline is mutable, and MachSMT has an interface to connect to any major ML platform. We describe each subsequently. First, all feature values are scaled to zero mean and unit variance[1]. This data normalization technique is common in ML research and applications to improve model efficiency and numerical robustness.

The second step in the preprocessing pipeline is computing the polynomial interaction terms of degree two on the normalized feature vector. These polynomial features make interacting correlations of features explicit. These first two preprocessing steps are included in the SATZilla preprocessing pipeline [202].

As discussed in Chapter 2.2, ML in a high-dimensional space is prone to the curse of dimensionality. While other algorithm selection tools (e.g., SATZilla) commonly implement feature selection solutions, we propose to use learnt dimensionality reduction, namely, PCA. As discussed above, feature selection can be a proactive solution to the curse of dimensionality but presents many challenges when applied to SMT. Internally, MachSMT manages up to more than a thousand learnt models (e.g., in the context of our SMT-COMP analysis in Section 3.3), and calculating optimal feature subsets for each one is infeasible.

The third and final preprocessing step is (optionally) applying PCA to the polynomial features. The final feature vector is composed of the first 35 principal components and is used to construct the learnt models with AdaBoost. This step can dramatically improve building times on larger data sets with many logics. Furthermore, several supervised learning techniques are more performant with PCA enabled. Concretely, MachSMT computes the 196+ features from the input, and the means and deviations are computed on each component in the feature vector across the provided data set. MachSMT then expands

[1] $\frac{x-\mu}{\sigma}$, where x is a feature sample, μ is the mean across the specific feature on the training set, and σ is the deviation across the specific feature on the training set.

this vector by computing polynomial interaction features of degree 2 (i.e., products of all components), resulting in an expanded feature vector of nearly 40,000 dimensions. Finally, we reduce this to a vector space of just 35 dimensions with PCA, which aims to preserve as much of the original information as possible from this large vector space. The reduction to 35 dimensions is a hyperparameter of PCA and MachSMT, and was empirically determined.

3.2.2 Supervised Learning Core

Depending on the quality and type of data available, different machine learning techniques may have different tradeoffs, as well as development and production constraints. For example, linear ridge regression is a popular regression algorithm for algorithm selection [202] with relatively very fast training times and very interpretable models. It is, however, not always the most performant technique. On tabular data problems, decision trees and AdaBoosting increase the overall accuracy of the learned models at the cost of interpretability and training times. Furthermore, it may be beneficial to deploy deep learning techniques in the presence of large amounts of data or significant resources.

MachSMT is written in Python and aims to be ML framework and algorithm agnostic. By default, MachSMT leverages scikit-learn [129] and numpy [79]. However, it is easy to extend MachSMT to support any ML model/pipeline under scikit-learn syntax, specifically, any class with `fit` and `predict` methods. Furthermore, it includes interfaces to other relevant frameworks, notably PyTorch [127] and XGBoost [48]. By default, MachSMT is configured to use AdaBoost with 200 decision tree estimators and linear loss as its supervised learning core. Alternatively, it can be configured to use linear ridge regression or a multi-layer perceptron (MLP) deep neural network.

3.2.3 Configurations of MachSMT

MachSMT implements the following algorithm selection solutions.

1. **Mach-LogicEHM** – This configuration of MachSMT is analogous to the algorithm selection approach taken by SATZilla, with an EHM is constructed for every solver/-logic pair. As state-of-the-art SMT solvers implement significantly different algorithms depending on the logic of the input problem, data points from different logics could negatively skew predictions.

Feature	Description	Feature	Description
1–4	Frequency of constructs for problem description (`as`, `assert`, `check-sat`, `check-sat-assuming`)	52–87	Frequency of constructs of the theory of bit-vectors (e.g., `BitVec`, `bvor`, `bvadd`, `bvult`, …)
5–13	Frequency of constructs for declaration and definition (e.g., `declare-const`, `define-fun`, …)	88–133	Frequency of constructs of the theory of floating-point arithmetic (e.g., `FloatingPoint`, `RNE`, `fp.add`, `fp.leq`, `fp.isNaN`, …)
14–15	Frequency of `echo` and `exit` constructs	134–148	Frequency of constructs of the theories of integers and reals (e.g., `Int`, `Real`, `>`, `+`, `*` `to_real`, `is_int`, …)
16–27	Frequency of `get-*` constructs (e.g., `get-model`, `get-unsat-core`, …)	149–184	Frequency of constructs of the theory of strings and regular expressions (e.g., `String`, `RegLan`, `str.len`, `str.++`, `re.diff`, …)
28–29	Frequency of `push` and `pop` constructs	185	Avg. number of `select`s per array
30–31	Frequency of `reset` and `reset-assertions` constructs	186	Avg. `store` chain depth per array
32–34	Frequency of `set-*` constructs (`set-info`, `set-logic`, `set-option`)	187–189	Avg./Median/Deviation of the depth of BV adder chains
35–36	Frequency of `forall` and `exists` bindings	190–192	Number and ratio of `forall`/`exists` variables
37	Frequency of `let` bindings	193	Avg. quantifier nesting level
38–48	Frequency of core and Boolean constructs (e.g., `true`, `Bool`, `and`, `or`, `=>`, `ite`, …)	194–195	Avg. arity and number of applications of uninterpreted functions
49–51	Frequency of constructs of the theory of arrays (`Array`, `select`, `store`)	196	Size of the smt2 file in bytes

Table 3.1: Complete list of the 196 features used in MachSMT.

2. **Mach-LogicPWC** – This configuration of MachSMT deploys the PairWise comparator approach as described in Chapter 3.2. In this configuration of the PWC, comparators are trained for every pair of solvers on the benchmarks of a common logic.

3. **Mach-Greedy** – In this configuration, the algorithm that performed the best over the training data is returned. Similar to Mach-LogicEHM and Mach-LogicPWC, it is filtered by the logic. This solution is a fallback for when MachSMT fails to learn a model that improves on the best standalone solver (as determined by the SMT-COMP results).

By default, MachSMT creates models for Mach-EHM and Mach-Greedy in the presence of one logic, and Mach-LogicEHM in case of multiple logics. In evaluation, MachSMT evaluates the performance of each approach on each logic. In deployment, MachSMT uses the approach that had the best-observed performance in evaluation.

3.2.4 Using MachSMT

MachSMT consists of three core tools, which are used to build, evaluate, and deploy MachSMT, respectively.

```
$ machsmt example.smt2 -l dir
 MachSMT selects: bitwuzla
      bitwuzla 81.102%
      cvc4 16.180%
      Z3 2.718%
```

Figure 3.2: Example Usage of MachSMT.

1. `machsmt_build` – This tool is the interface for building MachSMT's database around the solvers and benchmarks provided by the user. It takes as input a csv data file with columns *solver*, *benchmark*, and *score*. The solver column denotes the name of the solver, and the benchmark column indicates where the benchmark is located. We use `PAR-2` as score, with penalties for timeouts and incorrect answers (see Section 3.3.2). With default settings, the output of `machsmt_build` is a library directory containing the database and learnt models. For example, on input `data.csv` with output directory `dir`, `machsmt_build` is called as follows:

   ```
   $ machsmt_build -f data.csv -l dir
   ```

2. `machsmt_eval` – This tool takes as input the library directory with the database and learnt models generated by `machsmt_build`, evaluates them under k-fold cross validation, and provides a summary of the results. `machsmt_eval` further tunes MachSMT to use the best empirically observed configuration based on the logic of the input benchmark. For example, on `dir`, it is called as follows:

   ```
   $ machsmt_eval -l dir
   ```

3. `machsmt` – This tool is the primary interface to MachSMT's algorithm selection. Given an input benchmark and the library files generated above, `machsmt` will output a ranking of solvers, ordered from predicted to solve the input the fastest (top) to slowest (bottom). An example usage of `machsmt` on input file `example.smt2` with library directory `dir` is given in Figure 3.2.

3.2.5 Allocating Resources with MachSMT

In Section 3.2, we present an example command-line usage of MachSMT primary prediction interface. We suggest two different ways to allocate resources: either greedily spend all

resources on the top selected solver or allocate resources based on the suggested allocation (see Section 3.4 for a comparison).

MachSMT utilizes a resource allocation scheme that leverages ideas from deep neural networks—specifically, *softmin*, an output activation function that computes the smooth minimum of, in our case, a vector $\vec{x}$ of predicted solver runtimes. Leveraging EHMs yields a predicted runtime x_i for every solver $s_i \in \mathcal{S}$. The regression models can be poor at predicting absolute ground truth runtime accuracy, but the predicted runtimes of the solver EHMs relative to each other, however, can be indicative for algorithm selection. We thus chose to compute the suggested resource allocation for a solver s_i as a softmin of x_i, defined as follows.

$$softmin(x_i) = \frac{e^{-x_i}}{\sum_{j=1}^{n} e^{-x_j}}$$

The softmin activation function normalizes predicted solver runtimes to predicted probabilities, and is inspired by classification in ML, where the *softmax* output activation function is frequently used to compute a smooth maximum. In contrast to softmax, softmin allows us to give preference to lower runtimes.

3.2.6 User-defined Features

MachSMT provides a simple interface for users to extend its set of predefined features with user-defined features. Each new feature is represented as a Python method that returns a single floating-point number (or an iterable that returns floating-point numbers). Features can be disabled, by having the user-defined procedure return `None`. All user-defined features are automatically included in building MachSMT and can significantly affect the accuracy of MachSMT when engineered to target a specific class of benchmarks.

3.3 SMT-COMP

In this section, we present the evaluation of MachSMT, specifically on the benchmarks, solvers, and solver runtime analysis from SMT-COMP 2019 [78] and 2020 [17]. The results of this experiment are summarized in Tables 3.5, 3.4, 3.3, 3.2 and Figure 3.3.

3.3.1 Experimental Setup and Methodology

In this experiment, we used the benchmarks, timing analysis, and solvers provided by the organizers of the SMT-COMP 2019 and 2020 competitions [78, 17]. The competition is organized in tracks, which are split into divisions. In 2019 and 2020, as in the years before [184], a division corresponded to an SMT-LIB logic. We consider all divisions of the Single Query (SQ) and Incremental (INC) tracks for each year. In the SQ track, each input only contains a single `check-sat` query, whereas in the INC track, inputs contain multiple such queries.

In both years, all solver input queries were performed on the StarExec computing service [169], which consists of a cluster of 2.4 GHz Intel Xeon machines running Red Hat Enterprise Linux 7.2. Each solver/ benchmark pair was configured to have four cores and 60GB of memory available. The time limit for each pair was 2400 seconds in 2019, and 1200 seconds in 2020.

For this experiment, we configure MachSMT with AdaBoost with 200 decision tree estimators and linear loss as its supervised learning core. Further, we evaluate MachSMT in all of its configurations (as introduced in Section 3.2.3) using k-fold cross validation with $k = 5$. The data set is randomly partitioned into k subsets per division and year for cross-validation. For more details on cross-validation, see Section 2.2.

As we are leveraging data collected by the SMT-COMP organizers, we do not rerun with the allocation scheme as outlined in 3.2.5. Instead, we use whichever solver had the lowest predicted runtime. Note that this is usually slower than the allocation scheme.

3.3.2 Experimental Results

For each division, we compare MachSMT against the best solver of the division in the competition (including non-competing solvers). For the *Single Query* track, we evaluate solvers across, according to `PAR-2` scores [109]. Given the wallclock runtime w and the wallclock time limit t, we compute the `PAR-2` score as w on successful termination, $10 * t$ for an incorrect answer, and $2 * t$ for any other case of unsuccessful termination. For the *Incremental* track, we use $w + (2 * \frac{t}{n}) * (n - m)$, with n the total number of `check-sat` calls in the input and m the total number of `check-sat` calls successfully solved.

The results of this experiment are given in Tables 3.5, 3.4, 3.3, 3.2 and Figure 3.3. We consider all divisions that contain at least 25 benchmarks and exclude incremental benchmarks with only a single `check-sat` query. We further consider three baselines when evaluating MachSMT: random algorithm selection (Random), the best solver of the division

Logic in SQ'20	Best	PAR-2				
		Random	Best	VBS	Mach-LogicEHM	Impr. [%]
QF_BVFPLRA (153)	MathSAT5	48277.5	31111.1	159.5	**174.7**	**99.4**
LIA (299)	CVC4	393648.4	37071.9	2697.3	**12490.4**	**66.3**
BVFP (224)	CVC4	289843.2	266428.9	54434.9	**95958.1**	**64.0**
QF_UFBV (217)	Yices	35418.8	22343.0	5343.9	**9356.6**	**58.1**
UFDTNIRA (736)	Vampire	561429.7	177066.7	48076.2	**75373.3**	**57.4**
QF_LIA (2508)	CVC4	1246085.0	579953.4	169595.8	**269473.4**	**53.5**
BV (696)	CVC4	459663.8	239717.7	60146.9	**129250.2**	**46.1**
QF_UFNRA (27)	Yices	42957.1	12929.8	4670.0	**7902.6**	**38.9**
QF_ALIA (116)	Yices	39316.5	68.5	32.9	**45.8**	**33.2**
QF_NRA (2230)	Yices	1832304.9	1132073.9	468148.2	**812778.6**	**28.2**
AUFDTNIRA (300)	Vampire	268191.0	23710.6	5964.7	**17211.1**	**27.4**
QF_ABVFPLRA (41)	CVC4	15213.5	3269.6	30.0	**2436.3**	**25.5**
FP (993)	Z3	928925.1	462175.2	306571.7	**404901.1**	**12.4**
QF_IDL (834)	Yices	468813.2	234514.8	182994.3	**211312.8**	**9.9**
AUFDTLIRA (4878)	CVC4	4139948.8	1316952.2	1003100.6	**1189167.9**	**9.7**
UFDTLIRA (4296)	CVC4	4088170.6	1936992.8	1709161.2	**1769376.9**	**8.7**
QF_BV (6861)	Bitwuzla	1729172.7	431768.2	244154.8	**400073.9**	**7.3**
QF_ANIA (94)	MathSAT5	66033.4	617.9	251.1	**573.9**	**7.1**
UFNIA (5041)	Vampire	6813123.2	913033.2	603779.5	**861190.5**	**5.7**
AUFLIA (1310)	Vampire	1069749.1	507919.1	361485.5	**481708.6**	**5.2**
UFBV (72)	Z3	96086.8	55232.8	52830.0	**52832.7**	**4.3**
ABV (169)	Z3	357625.1	307762.2	295252.8	**297652.7**	**3.3**
QF_NIA (9195)	MathSAT5	10541986.4	7230607.6	4390297.4	**7024329.8**	**2.9**
AUFNIRA (300)	Vampire	627468.0	582179.9	549239.6	**566789.9**	**2.6**
QF_UFLIA (300)	Yices	74547.2	56.3	46.2	**54.9**	**2.5**
QF_RDL (247)	Yices	117274.8	83921.9	83164.3	**83598.5**	**0.4**
QF_UFIDL (300)	Yices	40814.8	5523.1	5490.4	**5508.2**	**0.3**
AUFLIRA (1346)	CVC4	796897.2	243525.3	228066.0	**242965.1**	**0.2**
FPLRA (27)	CVC4	55202.7	29740.3	29740.3	29740.3	0.0
BVFPLRA (185)	CVC4	338419.4	223227.4	223227.4	223227.4	0.0
NRA (93)	Z3	69642.8	7497.5	38.9	7498.0	0.0
ABVFP (30)	CVC4	40818.9	36003.5	36003.4	36003.5	0.0
UFFPDTLIRA (373)	CVC4	525984.4	103911.2	103911.2	103911.2	0.0
AUFBVDTLIA (683)	CVC4	1241067.5	791289.9	791289.9	791289.9	0.0
AUFFPDTLIRA (154)	CVC4	220808.6	48019.5	48019.5	48019.5	0.0
QF_UFLRA (432)	Yices	97099.5	4837.3	3287.3	4884.9	-1.0
QF_AX (300)	Yices	66061.7	6.3	5.3	6.5	-2.8
UFLIA (2278)	CVC4	3263274.9	2363827.7	2139934.2	2439073.8	-3.2
ABVFPLRA (75)	CVC4	158401.1	136806.3	136806.3	141602.0	-3.5
UFDT (1223)	CVC4	2296525.6	1990066.9	1889835.6	2059603.1	-3.5
QF_LRA (429)	OpenSMT	147136.2	71949.4	46349.1	76132.2	-5.8
UF (2291)	Vampire	4042624.6	3122146.8	2917476.7	3326149.0	-6.5
LRA (802)	Z3	788146.6	172873.6	71219.3	191840.8	-11.0
QF_BVFP (394)	Bitwuzla	20315.1	778.4	205.5	883.0	-13.4
QF_AUFBV (34)	Yices	42033.0	25818.9	20431.4	29298.1	-13.5
QF_UFNIA (300)	CVC4	152541.4	48058.9	12013.2	55528.3	-15.5
QF_FP (247)	Bitwuzla	134069.5	78980.8	42799.3	94053.4	-19.1
QF_ABV (3385)	Bitwuzla	126991.6	48643.1	42430.1	58977.6	-21.2
QF_SLIA (27033)	CVC4	4751221.5	698072.5	563735.7	889222.2	-27.4
AUFDTLIA (147)	CVC4	152649.5	23515.5	23255.2	31105.0	-32.3
QF_FPLRA (51)	COLIBRI	24654.4	7275.9	162.8	9663.8	-32.8
UFDTLIA (277)	CVC4	397736.7	250883.7	207326.4	355323.4	-41.6
QF_ABVFP (500)	Bitwuzla	66977.7	5460.8	3690.7	9589.7	-75.6
QF_UF (2800)	Yices	397239.9	388.8	325.8	724.5	-86.3
QF_S (927)	CVC4	69954.9	10415.2	4324.1	22949.6	-120.3
QF_AUFLIA (467)	Yices	228647.7	18.3	9.1	2412.6	-13058.7

Table 3.2: Results of Mach-LogicEHM on data from the SMT-COMP'20 SQ divisions.

Logic in INC'20	Best	PAR-2				
		Random	Best	VBS	Mach-LogicEHM	Impr. [%]
QF_ABV (376)	Yices	50028.7	12049.0	2283.9	**6493.4**	**46.1**
QF_AUFLIA (72)	Yices	24978.5	3293.0	2924.5	**2927.8**	**11.1**
QF_UFLIA (230)	Z3	158480.1	129340.4	107652.4	**117549.1**	**9.1**
UFLRA (748)	Z3	1131365.1	184002.0	159210.5	**174155.4**	**5.4**
UFNIA (2029)	Z3	3807401.9	2757190.6	1961676.0	2787434.9	-1.1
QF_LIA (69)	Yices	73247.5	45262.5	40039.5	45830.5	-1.3
QF_BVFP (51)	Bitwuzla	12529.7	371.3	89.1	379.5	-2.2
QF_UF (306)	Yices	2645.4	64.3	59.8	66.6	-3.5
UF (2031)	Z3	4034456.8	2391318.1	1692446.9	2612107.1	-9.2
AUFNIRA (165)	CVC4	146343.7	99766.7	60200.6	116514.8	-16.8
QF_AUFBV (30)	Yices	28754.0	14092.7	12687.6	17773.8	-26.1
QF_BV (484)	Bitwuzla	42345.7	20437.5	17019.1	25880.7	-26.6
QF_UFLRA (1223)	Z3	19339.8	8344.7	5198.8	10607.5	-27.1
QF_ALIA (44)	Z3	8597.9	702.3	398.0	1517.3	-116.1
QF_UFBV (932)	Bitwuzla	11434.8	1831.5	832.3	4574.3	-149.8

Table 3.3: Results of Mach-LogicEHM on data from the SMT-COMP'20 INC divisions.

Logic in INC'19	Best	PAR-2				
		Random	Best	VBS	Mach-LogicEHM	Impr. [%]
QF_UFLIA (302)	Z3	377725.6	343663.9	248671.6	**267783.4**	**22.1**
QF_ABV (469)	Yices	165761.3	13054.3	3727.9	**11461.8**	**12.2**
QF_UFLRA (1529)	Z3	33688.3	11937.7	7505.0	**10653.4**	**10.8**
QF_AUFLIA (72)	Yices	7915.0	3029.5	2901.4	**2936.0**	**3.1**
UFLRA (935)	Z3	3355019.8	223811.2	214824.6	**223677.4**	**0.1**
QF_UF (383)	Yices	341.2	78.2	78.0	78.2	0.0
QF_LIA (69)	Yices	120316.9	72456.8	68228.8	73274.1	-1.1
AUFNIRA (165)	CVC4	465185.6	183826.9	109727.3	200152.9	-8.9
QF_AUFBVNIA (32)	MathSAT	53475.0	849.6	667.0	1003.1	-18.1
QF_BV (583)	Boolector	270820.1	40616.1	29793.1	59064.0	-45.4
QF_UFBV (1165)	Boolector	14560.1	3261.0	638.1	7311.1	-124.2
QF_ALIA (44)	Z3	3440.9	854.5	394.1	5334.1	-524.3

Table 3.4: Results of Mach-LogicEHM on data from the SMT-COMP'19 SQ divisions.

Logic in SQ'19	Best	PAR-2				
		Random	Best	VBS	Mach-LogicEHM	Impr. [%]
QF_UFBV (223)	Yices	57238.9	26322.4	2536.0	**2891.0**	**89.0**
QF_UFNRA (26)	Yices	81791.6	20744.2	5635.6	**8119.3**	**60.9**
NRA (93)	Vampire	161246.9	53040.4	4891.9	**24018.8**	**54.7**
QF_NRA (2842)	Yices	4700323.3	3307748.9	918003.5	**1720350.2**	**48.0**
QF_FP (225)	COLIBRI	311524.5	252284.1	111486.3	**177289.9**	**29.7**
BV (823)	Q3B	775521.1	407336.7	129831.8	**294642.1**	**27.7**
QF_LIA (3136)	SPASS-SATT	3493282.0	493192.2	183339.1	**362941.9**	**26.4**
AUFLIRA (1683)	Z3	3302901.6	398577.1	316990.7	**327028.9**	**18.0**
QF_LRA (546)	SPASS-SATT	346010.5	128635.0	63374.3	**105550.3**	**17.9**
QF_AUFLIA (651)	Yices	528768.8	18.6	11.6	**15.3**	**17.5**
LIA (299)	Z3	749636.7	14.8	11.3	**12.9**	**13.1**
AUFLIA (1638)	Vampire	3423183.4	1242230.7	851466.2	**1104767.7**	**11.1**
UFNIA (6253)	CVC4	19759215.9	13320056.8	7804961.4	**11927375.3**	**10.5**
QF_IDL (1042)	Z3	1477558.6	557543.6	437743.1	**509547.6**	**8.6**
AUFNIRA (300)	CVC4	1302356.6	1171565.6	1110988.5	**1155968.5**	**1.3**
FP (1238)	Z3	2470540.0	1006063.1	719531.7	**1000203.6**	**0.6**
QF_UFNIA (300)	CVC4	260047.3	72279.0	19207.9	**72091.2**	**0.3**
QF_RDL (247)	Yices	182890.5	162609.5	162070.6	**162327.5**	**0.2**
UFDT (1547)	CVC4	5342330.5	5011100.6	4601262.4	**5004303.6**	**0.1**
QF_UFIDL (300)	Yices	62584.7	10206.4	10172.4	**10197.5**	**0.1**
QF_NIA (11494)	CVC4	28208367.0	17006943.3	10126265.9	17009089.1	0.0
UFBV (72)	Z3	179316.1	110434.2	100836.9	110438.1	0.0
QF_AX (300)	Yices	107798.6	4.8	4.6	4.8	-0.3
QF_AUFBV (41)	Yices	39919.8	14811.6	6250.6	14900.5	-0.6
QF_BVFP (516)	CVC4	25474.1	8997.5	3643.2	9063.2	-0.7
QF_UFLIA (300)	Yices	193160.2	31.6	19.9	32.0	-1.4
UF (2816)	Vampire	10822123.3	8052636.1	7371252.1	8180674.7	-1.6
UFLIA (2848)	CVC4	9383261.2	5857081.7	5426481.9	5967471.7	-1.9
QF_ABV (7538)	Poolector	236916.1	104633.3	84039.8	107399.8	-2.6
AUFDTLIA (275)	CVC4	191254.3	46638.2	23377.1	51228.3	-9.8
QF_BV (8909)	Poolector	4757776.7	868489.1	549558.4	1021521.2	-17.6
LRA (1003)	Z3	2022790.0	363124.4	223245.7	473842.6	-30.5
UFDTLIA (299)	CVC4	752499.4	495667.2	434393.8	725651.2	-46.4
QF_UFLRA (541)	SMTInterpol	280508.4	3214.4	1964.0	4972.3	-54.7
QF_UF (3512)	Yices	1098010.6	422.4	272.8	1451.2	-243.6
QF_ALIA (139)	Yices	156602.7	57.9	50.1	4852.9	-8274.6

Table 3.5: Results of Mach-LogicEHM on data from the SMT-COMP'19 INC divisions.

(Best), and the virtual best solver (VBS). Tables 3.5, 3.4, 3.3, 3.2 give the `PAR-2` score for each configuration, as well as the percent improvement of Mach-LogicEHM over the best solver of each division. We only report the results for Mach-LogicEHM as it is, overall, the best performing configuration. Note that the VBS corresponds to perfect algorithm selection and can therefore not be beaten. We observe that overall and across all four tracks, MachSMT improves on the best solver in 57 (out of 119) divisions— for SQ'20 on 28 (out of 56), for INC'20 on 4 (out of 15), for SQ'19 on 20 (out of 36), and for INC'19 on 5 (out of 12).

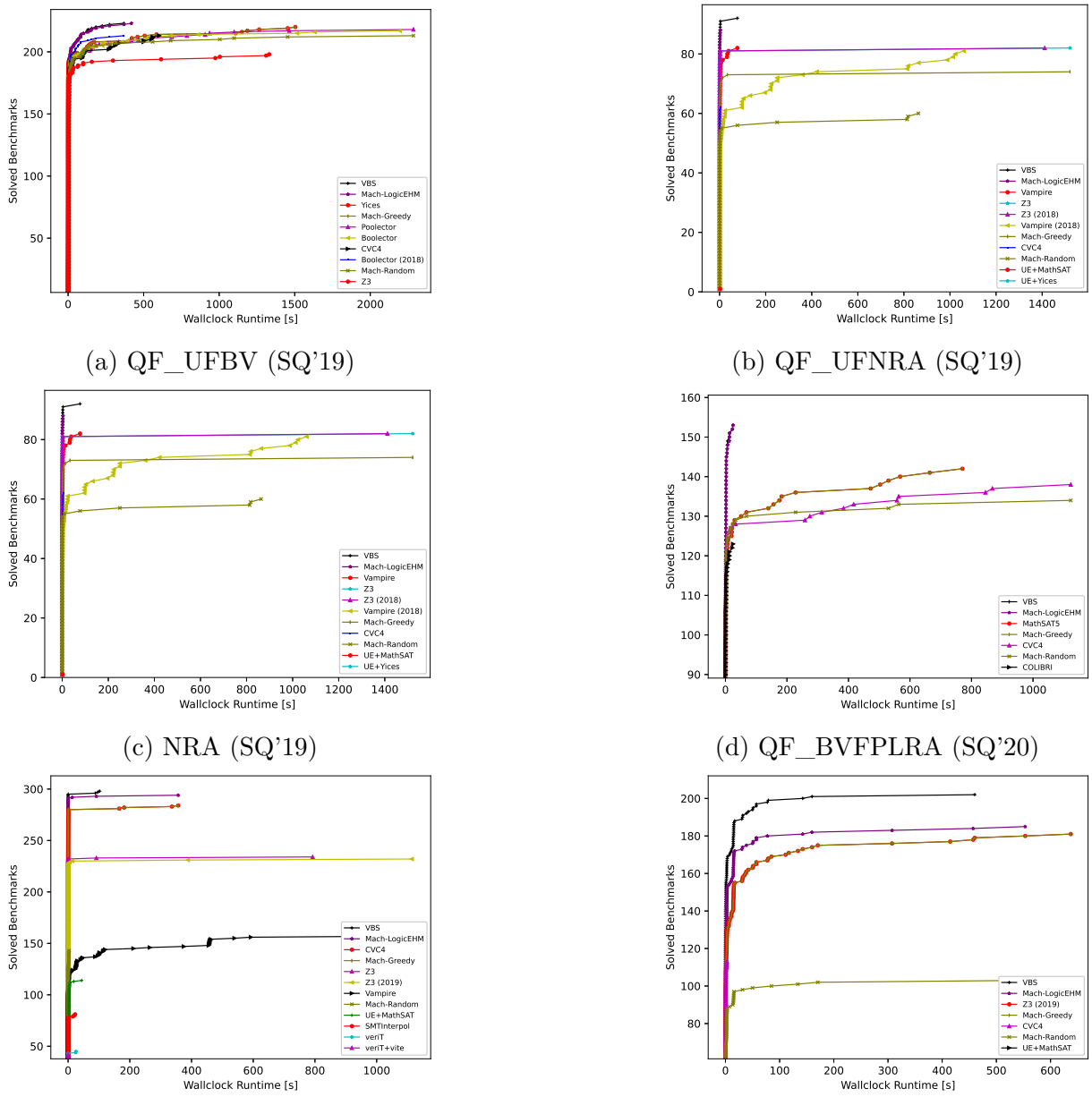
We present selected cumulative distribution function (CDF) plots of the top three results with the most improvement in the SQ'19 and SQ'20 tracks in Figure 3.3. In this context, a CDF visualizes how a solver performs on a database of inputs. A point (x,y) denotes that a solver $\mathcal{S}$ solves y inputs within x seconds each.

3.3.3 Discussion

In Section 3.2.3, we describe three configurations of MachSMT. In our evaluation of SMT-COMP data, we observe that Mach-LogicEHM performs significantly better than the other MachSMT configurations. Our experimental results confirm that algorithm selection (in particular through the use of EHMs) can be a powerful way to address the combinatorial explosion that solver users face when trying to decide which solver-configuration pair is best suited for their application.

We note that MachSMT is particularly powerful in the context of divisions with a diverse set of solving procedures, e.g., QF_BVFPLRA and QF_UFBV. Possible reasons for divisions where MachSMT performs worse than the best performing solver are too few (or too homogeneous benchmarks), lack of diversity of the algorithms to select, and the feature vector not sufficiently capturing the challenges of an input problem. For example, an interesting observation occurs in QF_BV logics of the SQ tracks, on a specific set of small benchmarks, a benchmark family originating from [123]. This benchmark family encodes bit-vector rewrites, which are trivial if a solver implements these specific rewrites. However, MachSMT is tricked into picking the competition winner Bitwuzla, even though it misses some of these rewrites and performs worse on some of these benchmarks than other solvers. This is mainly due to the fact that these benchmarks are very small in size, and Bitwuzla usually outperforms other solvers on instances of small input size.

As noted in previous work, algorithm selection methods work well for non-homogeneous benchmarks, especially where no single algorithm performs best across the board. EHMs



(a) QF_UFBV (SQ'19)

(b) QF_UFNRA (SQ'19)

(c) NRA (SQ'19)

(d) QF_BVFPLRA (SQ'20)

(e) LIA (SQ'20)

(f) QF_BVFP (SQ'20)

Figure 3.3: Comparison of MachSMT, the virtual best solver, and participating solvers in divisions of the SMT-COMP 2019 and SMT-COMP 2020 Single Query (SQ) Tracks.

Logic	PAR-2			Solved		
	cvc5	MachSMT	MachSMT–alloc	cvc5	MachSMT	MachSMT–alloc
ALIA (42)	**2406.7**	2691.3	2921.7	41	41	41
AUFBV (1522)	2689925.2	**2608537.5**	2608683.6	410	**448**	446
AUFBVDTLIA (1690)	2403641.5	**1961063.1**	1967667.0	716	**876**	873
AUFBVFP (57)	90215.3	86963.7	**85188.4**	20	21	**22**
AUFDTLIA (728)	79737.0	**5190.7**	5264.2	698	**728**	**728**
AUFLIA (3276)	1422835.2	**1353325.9**	1353915.0	2695	2724	**2726**
AUFLIRA (20011)	**626978.9**	755386.7	760748.0	19755	**19766**	**19766**
AUFNIRA (1480)	1016878.9	1016541.9	**1008664.9**	1058	1060	**1064**
UF (7590)	9420855.1	8576358.4	**8573813.0**	3744	4047	**4053**
UFBVLIA (208)	497350.9	**494424.3**	494424.5	1	**2**	**2**
UFDT (4569)	5333216.9	5090884.3	**5074775.5**	2380	2463	**2471**
UFDTLIA (327)	381207.6	215555.2	**204909.8**	183	238	**243**
UFFPDTLIRA (774)	192772.6	**182239.3**	187409.0	696	**701**	699
UFIDL (68)	24515.3	**24402.7**	24408.9	58	58	58
UFLIA (10127)	**6041256.6**	6115048.8	6077710.4	7625	7610	**7630**
UFNIA (13463)	13258581.0	13089905.7	**12933659.8**	7988	8067	**8144**
Total (65932)	43482374.7	41578519.5	**41364163.7**	48068	48850	**48966**

Table 3.6: Comparison of MachSMT, MachSMT–alloc and the cvc5 competition script on all evaluated logics.

are an effective way to distinguish between such algorithms and predict which one might perform the best on a given instance.

The best algorithm selection methods were based on EHMs. We found that Mach-LogicEHM outperformed Mach-LogicPWC by a median PAR-2 score improvement of 59.8%. Over Mach-Greedy, on the other hand, Mach-LogicEHM has a median PAR-2 score improvement of 11.8%.

One major threat to the validity of any ML solution is the generalizability of the learnt models on unseen data. Previous work has noted that a practical way to address this issue is to use $k-$fold cross-validation scheme [141, 110], thus motivating our use of this approach in our experiments. We further note that our evaluation of MachSMT on SMT-COMP data includes decades of runtime analysis and more than 100 GB of benchmarks spanning numerous applications, giving us greater confidence in the robustness of our results.

3.4 CVC5

In this section, we use MachSMT to predict a ranking and resource allocation for a set of configurations of a single solver to be used in a sequential portfolio setting.

3.4.1 Experimental Setup and Methodology

In this experiment, we evaluate MachSMT for logics with uninterpreted functions (UF) and quantifiers on 23 different configurations of the SMT solver cvc5 [16], as taken from its SMT-COMP'21 script[2]. We use all benchmarks available in these logics from the 2021 release of SMT-LIB [19], the number of benchmarks for each logic is given in parenthesis in Table 3.6.

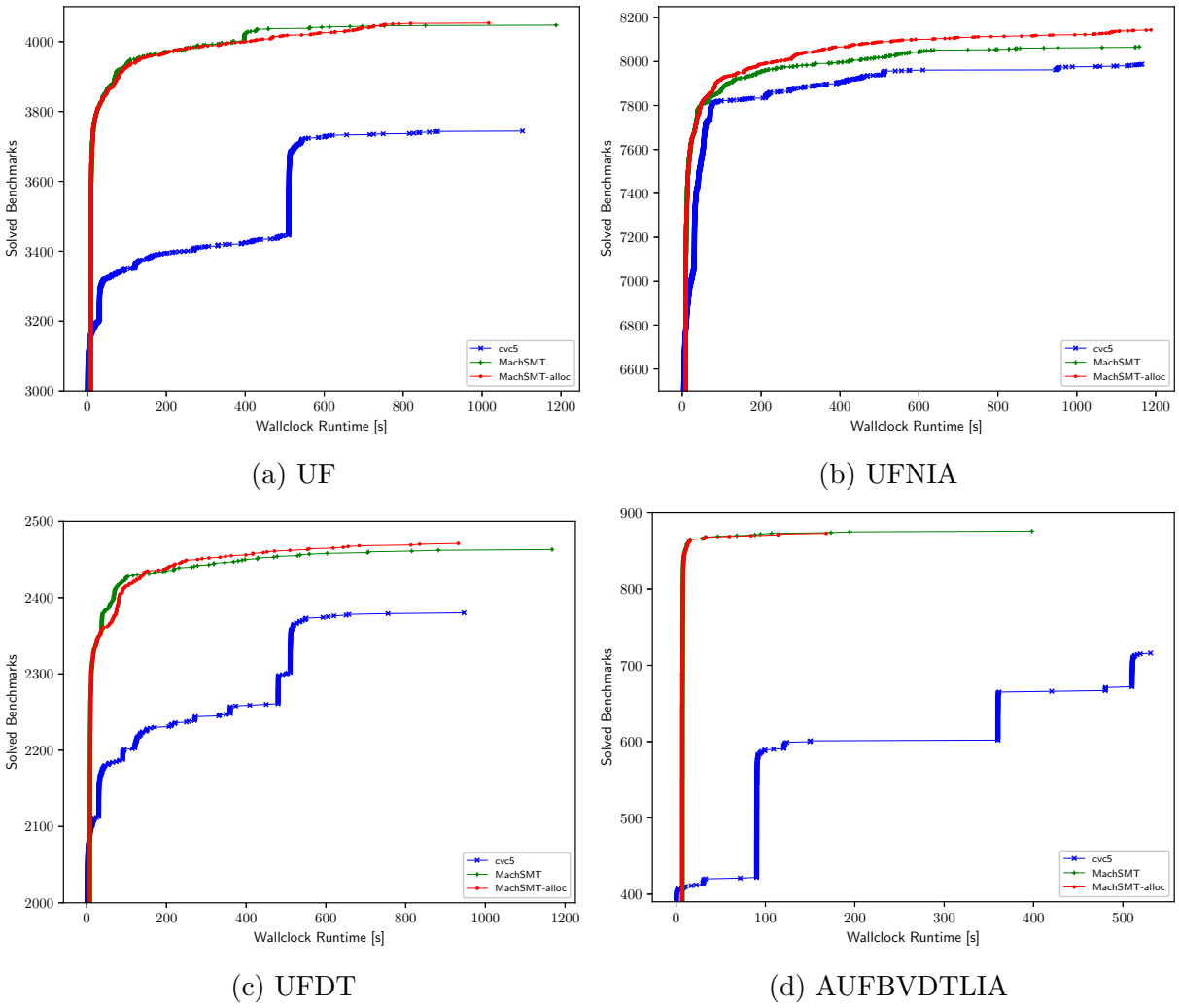
This experiment differs significantly from the experiment performed in Section 3.3 in the sense that all 23 configurations are configurations of a single solver—they only differ in their configured options, which enable or disable different algorithms and heuristics. In the competition setting, each of these configurations is ran sequentially, each with a specific time limit, until a solution is found or the overall time limit is exceeded. Resource allocations (the time limit) per configuration have been determined manually, and are configured based on the overall time limit of the competition. This sequential portfolio setting is encoded in the aforementioned competition script of cvc5.

Our goal for this experiment is to utilize MachSMT to predict a ranking and resource allocations for these configurations in a sequential setting that outperforms cvc5's competition script setting.

In Section 3.3, we evaluated MachSMT and all of its configurations (as introduced in Section 3.4) and observed that Mach-LogicEHM performs best. Hence, for this experiment, we exclusively consider this configuration. As discussed in Section 3.2, we suggest two resource allocation strategies based on the predictions produced by MachSMT. First, allocating all resources to the solver with the highest predicted performance. This was the only considered allocation strategy used in Section 3.3. Second, we use the resource allocation outlined in Section 3.2, where we compute a softmin over the predicted runtimes produced by the EHMs. We refer to these two configurations of MachSMT as *MachSMT* and *MachSMT–alloc*, respectively.

Our experimental setup is analogous to Section 3.3. However, for collecting the training data we ran all 23 solver configurations with a time limit of 60 seconds. We performed all experiments in this section on a cluster running Ubuntu 16.04 with Intel Xeon E5-2620 CPUs with 2.1GHz and 128GB memory. At runtime, for each solver/benchmark pair we used a time limit of 1200 seconds and a memory limit of 8GB. Further, we configure MachSMT's supervised learning core to train on an 8 layer multi-layer perceptron (MLP) neural

[2] https://github.com/cvc5/cvc5/blob/master/contrib/competitions/smt-comp/run-script-smtcomp2021



(a) UF

(b) UFNIA

(c) UFDT

(d) AUFBVDTLIA

Figure 3.4: Comparison of the two different MachSMT resource allocation schemes against cvc5 competition script.

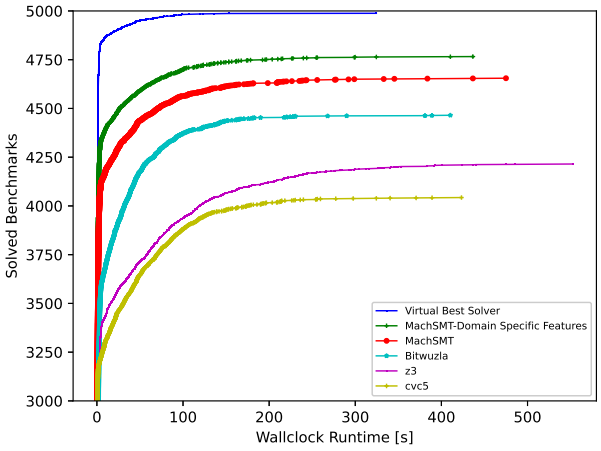


Figure 3.5: MachSMT with and without domain-specific knowledge on Network Verification Problems from SecGuru.

network of 128 neurons each. We use ReLU activation functions with batch normalization applied after each layer trained on an NVIDIA 2070 GPU for 30 minutes.

3.4.2 Experimental Results

The overall results of this experiment are presented in Table 3.6, which gives the `PAR-2` score and the number of solved instances for each configuration and considered logic. Figure 3.4 shows CDF plots of selected logics. In total, we observe that MachSMT-alloc solves 898 more benchmarks than the competition version of cvc5, and improves by 5% in total `PAR-2` score. Further, MachSMT is able to improve over the carefully hand-tuned sequential portfolio of the cvc5 competition script on the majority of the quantified logics—achieving a lower `PAR-2` score in 13 out of 16 logics while solving more benchmarks in 14 out of 16 logics with up to a 93.4% `PAR-2` improvement in the AUFDTLIA logic.

3.5 Network Evaluation

In this section, we present the evaluation of MachSMT on a network application, and show how domain specific knowledge can be utilized to extend MachSMT to further improve performance. The results of this experiment are summarized in Figure 3.5.

3.5.1 Experimental Setup and Methodology

In this section, we evaluate MachSMT's abilities to leverage basic domain specific knowledge to increase the overall prediction accuracy of learnt models and improve MachSMT's overall performance. For this experiment, we consider network verification problems, specifically, firewall and router queries based on the SecGuru tool by Microsoft Research [86].

We extend MachSMT via its user-defined feature interface (outlined in Section 3.2.6) to capture the basic attributes of the underlying query. Specifically, we include the following domain-specific features: categorical features denoting the benchmark type (firewall or router), number of rules, number of allow rules, number of deny rules, block rules, IP range width, and port range width. We train and test on two separate data sets of 5,000 SecGuru benchmarks.

We consider three underlying solvers: Bitwuzla [117], cvc5 [16], and Z3 [111]. All benchmarks produced are of the `QF_BV` logic. We baseline against MachSMT without the domain-specific features. Both variants of MachSMT are using Mach-LogicEHM mode for algorithm selection.

All experiments in this section were performed on the Compute Canada computing service [11], a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory with a wallclock runtime of 10 minutes. Wallclock runtimes are rounded to the nearest second.

This data set was collected offline, and we do not rerun with the allocation scheme as outlined in Section 3.2.5. Instead, we use whichever solver had the lowest predicted runtime. Note, however, that this is usually slower than the allocation scheme. In this experiment, we configure MachSMT's supervised learning core to use linear ridge regression with cross-validation on the regularization parameter[3].

3.5.2 Experimental Results

The results of this experiment are presented as a CDF plot in Figure 3.5. Overall, we observe MachSMT to improve on the best standalone solver Bitwuzla by 42.5% without the domain-specific features and by 77.3% with domain-specific features in `PAR-2` score. The only difference between the two configurations of MachSMT is the inclusion of the additional features—the domain-specific features account for 34.8% of the total improvement despite their simplicity.

[3] sklearn.linear_model.RidgeCV

3.6 Analysis

One major threat to the validity of any ML solution is the generalizability of the learnt models on unseen data. In the context of MachSMT, this scenario arises when asking MachSMT to solve classes of benchmarks that are divergent from the ones it has been trained on. In particular, in Section 3.3 we deployed $k-$fold cross-validation, which partitions the training and testing set. This scheme and its impacts on accuracy, bias, and error have been studied extensively [141, 110].

Furthermore, logics in SMT-LIB are organized in families, depending on the source and application of the benchmarks. Some families contain a small number of benchmarks, and some logics contain a small number of families. Both scenarios may result in MachSMT underperforming on a new (or underrepresented) benchmark set when using the models built on SMT-COMP data. Generally, when similar data is not available or underrepresented in the SMT-COMP models, users are strongly encouraged to include runtime data for their new benchmarks and rebuild MachSMT. Note that algorithm selection in the context of floating-point problems has been shown to generalize on industrial benchmarks from an independently fuzzed, synthetic data set [156]. This may be an alternative approach in cases where not enough benchmarks are available. Alternatively, it may be beneficial to merge related logics (similarly to what has been done in SMT-COMP to define divisions since 2021) to achieve larger data sets.

Determining feature importance to improve the efficiency of ML is an interesting and challenging question, MachSMT included. We are aware of two existing strategies: black-box analysis on the input-output behavior (e.g., SHAP [165]). Another way is to leverage model-specific attributes if possible. We do not consider the former in this chapter, and the latter is only applicable to certain types of models.

In this chapter, we considered 3 types of ML models: AdaBoosting (Section 3.3), multi-layer perceptron (MLP) neural network (Section 3.4), and Linear Ridge regression (Section 3.5). However, only the latter has a way to extract feature importance directly (via the weights). In this experiment, we noticed that the weights of the trained model were significantly higher for the domain-specific features, in particular the number of allowable rules and bit-vector multiplication.

By far, the best performing configuration of MachSMT is Mach-LogicEHM. Notably, Mach-LogicEHM outperformed Mach-LogicPWC and Mach-Greedy by a median `PAR-2` score improvement of 59.8% and 11.8%. From a regression standpoint, the overall model accuracy of precisely predicting the runtime for a benchmark is quite poor. However, even though individual predictions might not reflect the ground truth runtime, the relative

ranking of the predicted runtimes can be highly accurate, as observed in the chapter.

Limitations. One major limitation of MachSMT is its ability to leverage the provided data. As MachSMT builds its EHMs logic-wise, despite the size of SMT-LIB, several logics are very small with very few samples $N < 25$. Supervised learning is best suited in the presence of vast amounts of data, but when samples are scarce, MachSMT will likely perform poorly compared to standalone solvers.

Moreover, in the presence of broad data, which leads to highly variant solver performances, MachSMT is much more likely to improve on the best standalone solver. However, when the gap between VBS and the best standalone solver is minimal, MachSMT will most likely perform poorly. This is due to VBS, which, by its definition, serves as an oracle for blackbox algorithm selection, bounding MachSMT's performance.

Another major weakness of MachSMT is in its feature vector. In this chapter, we presented a very broad multipurpose feature vector that encodes relevant information across multiple problem instances. However, substantial amounts of information may be lost. In Chapter 3.5 we demonstrated how to help circumvent this when the domain is fixed. However, with improvements in deep learning, perhaps more raw representations of SMT formulae could be learned (e.g., through GPTs or graph neural networks) for algorithm selection.

Lastly, MachSMT is a blackbox and offline algorithm selector, i.e., it treats the solvers as a blackbox and does not leverage any of the online information that organically comes from the solving process. Oftentimes, this information can be extremely EHM predictions [198]. We leave this for future work and explore it in more context in Chapter 4.

3.7 Related Work

In this section, we provide an overview of previous work on algorithm selection in the context of constraint solvers and contrast it with MachSMT.

3.7.1 Key differences between SATZilla and MachSMT

As mentioned above, SATZilla was the first algorithm selection method in the context of logic solvers [198]. While SATZilla inspires our work, MachSMT differs from SATZilla in several key ways. First, SATZilla deploys a feature selection scheme to avoid the curse of dimensionality. While good in practice in the SAT setting, feature selection does lose a

significant amount of information. Further, it can be costly to compute optimal feature subsets.

In contrast, MachSMT leverages a learnt dimensionality reduction scheme, namely, Principal Component Analysis (PCA). The key advantage of PCA is that it does not perform a search for an optimal feature subset (as is required in the context of feature selection) and is thus significantly more efficient. Further, MachSMT deploys a modern ML pipeline, including an ensemble learning approach, namely Adaptive Boosting [61].

3.7.2 Algorithm Selection for Logic Solvers and Their Applications

Algorithm selection tools have a rich history and have been around at least since 1976, when Rice was the first to propose the technique [137]. Algorithm selectors have been extensively used in many contexts, e.g., in classifiers for machine learning [3], combinatorics [94], and other NP-hard optimization problems [175, 180, 124]. Within the context of constraint solvers, algorithm selectors have been proposed for QBF [133, 104], SAT [198, 199, 201], CSP solvers [66, 5, 85, 96], and recommenders for ATP tools [170, 179].

In the setting of SMT solver applications, symbolic execution tools have used algorithm selection strategies [186] and portfolio strategies [82] for the specific classes of instances within the context of the bit-vector theory. This would be an ideal use case of MachSMT since we provide a complete solution.

There have been other works using machine learning to improve the performance of SMT solvers. Balunovic et al. [13] use neural networks and synthesis to find tactics and strategies for three SMT-LIB theories. A previous version of our work proposed an algorithm selection tool for the QF_FP theory [156]. To the best of our knowledge, MachSMT is the first publicly available tool for the entirety of SMT-LIB. Other works have leveraged machine learning to improve internal heuristics in solvers [27, 139, 101].

Pairwise ranking has been used in algorithm selection in the latest versions of SATZilla [197] and in variable selection in the context of splitting heuristics in divide-and-conquer parallel SAT solvers [114].

Recently, Pimpalkhare et al. released a system for dynamic algorithm selection on SMT solvers [131]. Their chapter directly compares against MachSMT over four logics considered in Section 3.3. The advantage of a system like this is when runtime data is unavailable, and data must be collected online. However, their reinforcement learning approach still requires significant exploration to be predictive, and, in principle, MachSMT could also leverage

the runtime data they collect. Nevertheless, in their evaluation, MachSMT outperformed their system when runtime data was provided.

3.8 Conclusions

In this chapter, we presented MachSMT, the first algorithm selection tool that spans the entirety of the SMT-LIB logics. MachSMT is designed to be user-friendly and easily modifiable by users for their specific application and SMT solvers of interest.

We extensively evaluated MachSMT over several usage scenarios and empirically demonstrated its efficiency and efficacy. Using MachSMT, we observe improvement in 57 out of 119 divisions in all tracks from the SMT-COMP '19 and '20, with up to a 99.4% improvement in `PAR-2` score over the best performing solver for the QF_BVFPLRA SQ'20 division.

We further evaluated MachSMT to predict a ranking and resource allocation for 23 configurations used in the cvc5 competition script and observed that MachSMT was able to solve 898 more benchmarks with up to an 93.4% improvement in `PAR-2` score. Finally, we evaluated MachSMT on network verification problems with simple domain-specific knowledge and observed an improvement of 77.3% in `PAR-2` score.

For future work, we plan to extend our feature set with more (theory-)specific features based on feedback from the SMT community. Recent research on online data collection strategies has increased the usability and pragmatism of algorithm selection [131]. Furthermore, the feature vector in MachSMT has significant room for improvement. One up-and-coming candidate for a highly representative feature vector is using learnt features (i.e., graph neural networks) over the graphical syntactical structure of the logic of the problem at hand. This was recently studied by Hula et al. [84] with promising results.

Chapter 4

Meta-Solving for Neural Network Verification

In recent years we have witnessed a significant rise in interest in the Verification of Neural Network (VNN) problem, resulting in the development of a variety of complete and incomplete solvers that draw on diverse techniques. As is typical for hard search problems, no single solver is expected to be the fastest on all inputs. This insight suggests the use of algorithm selection techniques that automatically select the fastest solver for a given input.

Inspired by the success of algorithm selection for SAT and SMT solvers, we present `Goose`, an adaptive algorithm selection tool, which we dub a *meta-solver*, for deep neural network verification. Traditionally, algorithm selection tools have tended to be non-adaptive (i.e., they go from formula features to a solver). By contrast, a key novelty of our method is that `Goose` implements an *adaptive* sequential portfolio, i.e., it calls a set of subsolvers in a sequence, wherein the order in which subsolvers are called is determined adaptively based on their online and offline performance histories. We have implemented a variety of complete and incomplete subsolvers in `Goose` (in addition to using a set of off-the-shelf ones), and the following synergizing techniques to implement its adaptive sequential portfolio: algorithm selection, probabilistic satisfiability inference, and time-iterative deepening.

We evaluate `Goose` on the VNN-COMP '21 and VNN-COMP '22 benchmarks. We observe a 37.7% improvement over the competition winner in `PAR-2` score across nearly 900 benchmarks and 13 solvers from the competition in 2021. Furthermore, we observe

a 25.6% improvement over the competition winner in `PAR-2` score across more than 1500 benchmarks and 11 solvers from the competition in 2022.

4.1 Motivation

Over the last few decades machine learning (ML) has dramatically impacted many fields ranging from computer vision [98], natural language processing [182], and reinforcement learning based game playing [161]. ML systems are so ubiquitous today that one finds them even in safety-critical systems such as self-driving vehicles and drones [107], where one typically expects to find sub-systems whose robustness properties can be formally defined and checked using automated verification tools. While researchers have made considerable progress in developing verification tools for hardware and software systems, the field of formal verification of neural networks is still in its infancy.

The Neural Network Verification Problem. Consider the well-studied adversarial (or local) robustness problem for a trained neural network (NN). This problem is best illustrated with a simple example. Consider a trained NN designed to recognize pictures of stop signs and further assume that it has very high accuracy on a suitable test set. Researchers have figured out ways to perturb images of stop signs in subtle ways such that the perturbed image is classified as a green light, even though it is clear to a typical driver that the image is that of a stop sign [164].

Neural Network Verification Tools. In response to the above-mentioned adversarial robustness problems and despite the fact that this problem has been shown to be NP-complete [90], researchers in recent years have developed many solvers[1] that take as input an NN (in symbolic form), a robustness property, and output either an adversarial NN input or a guarantee that no such input exists [90, 145, 163, 177, 183, 194, 195, 207]. Over the years, a large variety of NN verification tools have been developed that can be broadly classified as complete solvers (e.g., either produce an adversarial input or provide a guarantee no such input exists or timeout) or incomplete solvers (e.g., either produce an adversarial input or unknown or timeout).

Complete solvers may use a variety of techniques ranging from lazy unwinding of ReLU activations within the simplex algorithm as in ReluPlex [90], a bound over-approximation algorithm on starsets [9] as in `nnenum`, all the way to direct translation of the problem

[1]The term solver refers to any computer program that takes as input a mathematical formula, and decides whether it has a solution.

into a Satisfiability Modulo Theories (SMT) or Mixed Integer Linear Program (MILP) [83, 177].

Similarly, incomplete solvers also use a variety of methods such as one based on abstraction-refinement [163], auto_LiRPA [183, 195], projected gradient descent (PGD) [50, 76], and fuzz testing [113]. Finally, some of the most successful solvers, such as α, β-CROWN, use many of the above techniques in a sequential portfolio[2].

Motivation for Algorithm and Adaptive Sequential Portfolio Selection. It is widely believed that no single algorithm is expected to perform well on all instances of an NP-hard problem. This means that a user of an NN verification tool cannot rely on a single solver to solve the kinds of problem instances they may be interested in. Unfortunately, manually figuring out which solver works best for a given instance can be time-consuming and laborious for any solver user. This is neatly summarized in the slogan "There ain't no such thing as free lunch" for hard search and optimization problems [192].

To address the above-mentioned problem in the context of solvers, researchers have proposed the use of the powerful paradigm of algorithm selection [138], which are functions that take as input a formula and output the name of the most efficient algorithm for the given formula. While a human can design such algorithms, it is very natural to use machine learning techniques to train a model that takes as input formula features and outputs the name of the most efficient solver for the given input formula. This technique was popularized by the ML-based algorithm selection SAT solver SATZilla [201] and recently for SMT solvers by the MachSMT [151] and Medley solvers [132].

Another approach that has been proposed is often referred to as the *sequential portfolio* of solvers, wherein a set of solvers is run on a given benchmark in a certain sequential order. For example, consider the competition-winning variant of the cvc5 SMT solver[3]. On just a single logic (e.g., UF), 23 configurations of the cvc5 tool are used, in a statically ordered sequential portfolio, to determine the satisfiability of the given benchmark.

While algorithm selection tools such as SATZilla [201] and MachSMT [151] have been shown to improve performance over any single solver dramatically, the non-adaptive nature of the algorithm selection used by them is limiting in many ways. For example, it is well known that solvers generate a lot of meaningful data at run time, as they make progress in solving an instance. Therefore it is natural to wonder whether this data can be used adaptively to improve on the initial choice of solver made by an algorithm selection tool.

[2] a term commonly used in the SMT solver context to denote a series of incomplete and complete techniques invoked in sequence

[3] https://github.com/cvc5/cvc5/blob/smtcomp2021/contrib/competitions/smt-comp/run-script-smtcomp-current

Problem Statement. The question we address in this chapter is the following: given a specific instance, is it possible to design an efficient adaptive ML-based sequential portfolio tool that dynamically changes its selections, as it solves an instance by leveraging online and offline performance histories of the underlying solvers on any given instance and does so by outperforming any single solver used as part of the portfolio.

To address the above-mentioned problem, we present `Goose`, a *meta-solver* for the verification of neural networks. By the term *meta-solver* we mean a tool containing a set of *subsolvers* that get adaptively called, in a sequence, based on online and offline information collected about their performance histories on a given input. This differs from traditional non-adaptive algorithm (resp. sequential portfolio) selection, i.e., once a solver (resp. sequential portfolio) is selected, that choice is not altered any further by the selection tool.

`Goose` implements three synergistic techniques. First is an ML-based *algorithm selection* technique that uses pre-trained empirical models for runtime prediction to construct an appropriate sequential portfolio of solvers, and allocate resources among the different solvers. Second, we construct ML models for *probabilistic satisfiability inference*. More specifically, `Goose` leverages a recent result [160] that reduces a given neural network verification input to a disjunction over a set of subproblems. Given this syntactic structure, we use ML-driven empirical models to predict satisfiability and rank subproblems based on their likelihood of being satisfiable. The rich structure of neural network verification problems allows for the engineering of expressive subproblem features; this contrasts with more generic SAT/SMT benchmarks, where the originating domain may not be known. We show that attempting to solve the subproblems in an empirically-determined appropriate order can greatly influence solving times. Third and last, we use *time iterative deepening*, an exponentially increasing wall-clock timeout on a sequential portfolio. Two insights underlie this strategy: (a) trying out multiple solvers with short time limits first is a good time-saving strategy, assuming that there exists a solver in the portfolio that can quickly solve the subproblem; (b) the failure of all solvers within a given time limit provides valuable information on the difficulty of a subproblem, which then *adaptively* informs the next round of algorithm selection, under a larger time limit.

4.2 Preliminaries

Empirical Models. An empirical model (EM) is a function learned from data that takes as input an encoding of a problem instance and is trained to fit ground truth labels collected from running the algorithm in question (e.g., runtimes, `PAR-2` score, `SAT/UNSAT`).

Empirical models are a generalization of Empirical Hardness Models (EHMs) in that they predict not only runtimes but also other aspects of a solver's behavior on a given instance, such as whether the input SAT/UNSAT and their `PAR-2` scores.

Solvers vs. Subsolvers: We sometimes use the term subsolver to simply refer to a solver, such as Marabou, selected by `Goose`. We do so only because we often refer to `Goose` as a meta-solver.

Solver Portfolios. The term solver portfolio has acquired multiple meanings within the solver community. In the parallel solver setting, a portfolio solver refers to a variety of *subsolvers* running in parallel on multiple nodes of a multiprocessor or a cluster [35]. In the sequential solver setting (that is relevant to this chapter), the term *(sequential) solver portfolio* refers to an ordered list of solvers and a time allocation for each as a fraction of a total time budget.

The Open Neural Network Exchange (ONNX). is a research and industry initiative to standardize machine learning models for the purpose of verification, analysis, and testing [57]. In the latest version (V18), the syntax and semantics of computation graphs are outlined for 187 operations (e.g., ReLU). ONNX is widely supported across major deep learning platforms such as TensorFlow [1], PyTorch [126], and Keras [51].

The `.onnx` file format for a *computation graph* has become the standard within the neural network verification community. A *computation graph* C is an abstract syntax-tree-like data structure that embeds the control flow of a machine learning program or model. A node $n \in C$, is made of an operator ϕ, a set of inputs X, and a set of outputs Y. While computation graphs are often exclusively considered in the context of deep learning, their expressibility is much richer and can also include models such as decision trees.

The International Benchmarks Standard for the Verification of Neural Networks (VNN-LIB). VNN-LIB is an international initiative to support neural network verification research [73]. In deep neural network verification, a verification query requires two parts: a computation graph C and a specification ψ. For the former, VNN-LIB defines syntax and semantics leveraging a subset of 17 ONNX operations, with the latest competition benchmarks leveraging 15 operators.

Problem Representations. For a computation graph C and a specification Ψ, the NNV question asks if Ψ is valid over C (i.e., $C \models \Psi$). Not all of the recently developed verifiers are compliant with arbitrary linear properties. For example, several solvers do not support disjunctions, conjunctions, nor linear real/integer constraints over the input/output. To overcome the above-mentioned problem, we leverage the following recent result:

Theorem 1 ([160]). *Let C be a computation graph and let Ψ be a linear specification over the input/output behavior of C (disjunctions, conjunctions, negations, and linear constraints). Then there exists subproblems ψ_i such that each ψ_i is of the form*

$$x \in \mathcal{X}' \wedge y \in \mathcal{Y}' \wedge Ay \leq b$$

for some intervals $\mathcal{X}', \mathcal{Y}'$, matrix A, vector b, and input/output x/y and

$$C \models \psi \iff \bigvee \psi_i \quad \textit{is} \quad \texttt{UNSAT}.$$

One of the most studied specifications in neural network verification is adversarial or local robustness in computer vision. The question asks, for a given image and a predicted class, whether a bounded perturbation of the original image exists that changes its predicted class. The syntactic substructure of ψ_i can be interpreted as a generalization of a local robustness query. Specifically, the bounds of $\mathcal{X}$ are computed with the middle interval point of the image at hand, with lower/upper bounds formed from the perturbation budget and the change of classification of the image with the halfspace-polytope constraint ($Ay \leq b$). This theorem allows us to construct a transformer $\mathcal{T}$ of the original problem to create an equivalent disjunction. We leverage this in multiple ways within `Goose`. A consequence, however, is a worst-case exponential blowup in the number of disjunctions. However, we note that we have not encountered this worst-case scenario in our experiments.

4.3 Goose

4.3.1 A High-Level Overview of `Goose`

`Goose` takes as input a computation graph C (in ONNX format) and a specification ψ (in VNN-LIB format), and outputs either VIOLATED/SAT (with a counter-example) or SAFE/UNSAT. `Goose` leverages an empirical model E to make predictions on runtimes (used in algorithm selection [4]) and probabilities of satisfiability (used in probabilistic satisfiability inference). `Goose` uses machine learning, specifically supervised learning, to realize E. This model is trained offline and requires data collection. We elaborate on this process in Section 4.4. In what follows in this Section, we assume the empirical model has already been trained.

[4] We use the terms algorithm selection and (sequential) portfolio selection interchangeably.

Feature	**Description**
Neural Network Encoding	
1	`.onnx` file size
2 – 4	Network Input/Output shape
5 – 27	Globally – Number of parameters, max, min, mean, and median, and percentile parameters at 5% increments
28 – 50	Gemm –Number of parameters, max min, mean, and median, and percentile parameters at 5% increments
51 – 73	Add/Sub Operator – Number of parameters, max, min, mean, and median, and percentile parameters at 5% increments
74 – 96	Constant operator – Number of parameters, max, min, mean, and median, and percentile parameters at 5% increments
97 – 119	Mul/Div operator –Number of parameters, max, min, mean, and median, and percentile parameters at 5% increments
120 – 142	Conv operator – Number of parameters, max, min, mean, and median, and percentile parameters at 5% increments
143 – 150	Number of Reshape, Pad, Gemm, Add, Sigmoid, Constant, Div and AveragePool nodes
151 – 157	Number of MatMul, Conv, Sub, Relu, MaxPool, Transpose, and Flatten nodes

Table 4.1: Neural Network Features

Feature	**Description**
Specification Encoding	
158	Number of assertions
159	Number of comparators (e.g., `<=`, `>=`)
160-162	Number of boolean operators and, or, not
163	Number of subproblems generated from $\mathcal{T}$
164	`.vnnlib` File Size
Online, Probing, and subproblem Encoding	
158	Known to not to be solved 'instantly' (Marabou) (1/0)
159	Known to not to be solved .5s (Marabou) (1/0)
160	Known to take at least 2s to solve (Marabou) (1/0)
161	Known to take at least 10s to solve (Marabou) (1/0)
162	Known to take at least 30s to solve (Marabou) (1/0)
163	Known to not to be solved 'instantly' (Complete + Gurobi) (1/0)
164	Known to not to be solved .5s (Complete + Gurobi) (1/0)
165	Known to take at least 2s to solve (Complete + Gurobi) (1/0)
166	Known to take at least 10s to solve (Complete + Gurobi) (1/0)
167	Known to take at least 30s to solve (Complete + Gurobi) (1/0)
168	Known to take at least 30s to solve (Complete + Gurobi) (1/0)
169	Known to not to be solved 'instantly' (PGD) (1/0)
170	Known to not to be solved .5s (PGD) (1/0)
180	Known to take at least 2s to solve (PGD) (1/0)
181	Known to take at least 10s to solve (PGD) (1/0)
182	Known to take at least 30s to solve (PGD) (1/0)
182	Known to take at least 30s to solve (PGD) (1/0)
183-188	PGD (in Hz) probing statistics (1/0)
188-197	Complete + Gurobi (in Hz) probing statistics (1/0)
198	Unstable Neuron fraction
199–221	Subproblem y pnorm distances from original network – max, min, mean, and median, and percentile parameters at 5% increments

Table 4.2: Specification and Encoding Features

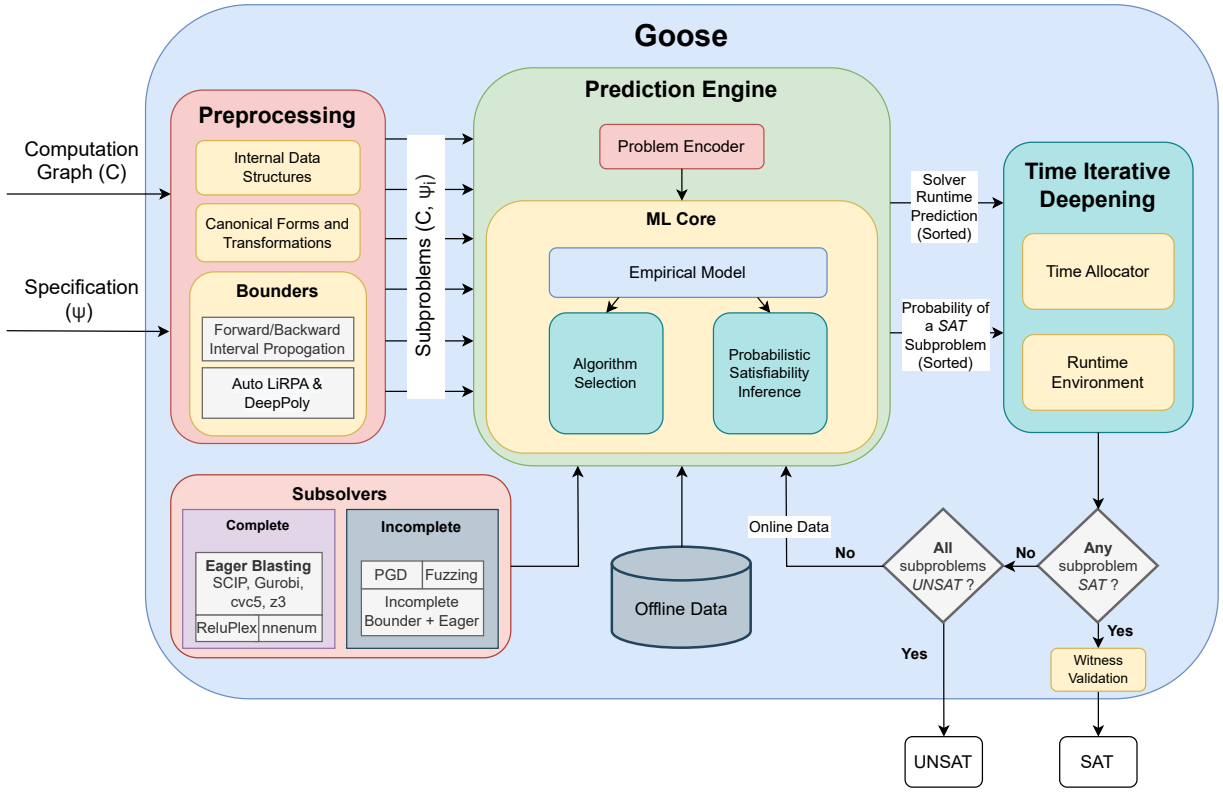


Figure 4.1: Architecture Diagram of `Goose` (See description in Section 4.3).

As stated above, the input to `Goose` is a computation graph C – a symbolic representation of the deep neural network of interest – and a linear specification ψ over the input/output behavior of C over all $x \in \mathcal{X}$. `Goose` outputs `UNSAT` if and only if $\forall x \in \mathcal{X}, C(x) \models \psi$.

At the start of its run, `Goose` loads the input into its internal data structures and converts them into a disjunction of subproblems ψ_i in a canonical form (Theorem 1). Next, `Goose` featurizes the input and leverages an offline trained empirical model to make predictions on how `Goose`'s subsolvers would perform on the problem at hand. Specifically, with the predictions from the model, `Goose` uses *algorithm selection* to determine which solvers should be used and *probabilistic satisfiability inference* to determine which subproblems to be targeted. We refer to the aforementioned modules of `Goose` as the "Prediction Engine".

`Goose` creates a solver portfolio for every subproblem ψ_i, leveraging algorithm selection. The time fraction for each subproblem's portfolio is determined using probabilistic satisfiability inference engine. The wall-clock timeout of all subproblem portfolios starts out very

small (e.g., ten seconds). `Goose` employs a *time iterative deepening* technique, inspired by the AI algorithm *Incremental Deepening* for two-player games [135]. The intuition is that in practice, runtimes of most solvers are usually either very short or long. Hence, when running a sequential portfolio of algorithms, we start with a shorter wall-clock portfolio timeout and exponentially increase it if the previously selected solvers in the portfolio fail to solve the input.

The problem encoder of `Goose` is designed in such a way that the encoding may change after an iteration of time iterative deepening. This allows us to leverage information collected from running the solvers at the end of an iteration, creating a feedback loop. `Goose` realizes a meta-solver as its ability to leverage the online and offline information in constructing solver portfolios for subproblems with algorithm selection and probabilistic satisfiability inference, adaptively, from running the solvers with time iterative deepening loop.

4.3.2 Input/Output and Preprocessing

Input/Output. As stated above, the input to the `Goose` system is a computation graph C (in `.onnx` format) and a specification over its input-output behaviour ψ (in `.vnnlib` format). We assume ψ restricts the domain and codomain of C to be bounded in $\mathcal{X}$ and $\mathcal{Y}$ (e.g., $0 \leq x_i, y_i \leq 1$). On a successful run, `Goose` outputs `UNSAT` if and only if $\forall x \in \mathcal{X}, C(x) \models \psi$, otherwise `SAT`. On a `SAT` result, `Goose` validates the *witness* – an assignment v to values of x, the input to the computation graph C. We validate the witness v by feeding it to C, computing the output $y = C(v)$, and checking that $\psi(v, y)$ evaluates to true.

Internal Data Structures. `Goose` has two key preprocessing steps. First, the inputs C and ψ are parsed and loaded into a graph and specification classes. These classes were designed to have significant utility to be later leveraged by a decision procedure. These data structures were designed to behave well with subproblem generation, bounding, and avoiding reparsing the input.

Canonical Form. The second step is the conversion of the input problem into a canonical form. Specifically, we implement a transformer $\mathcal{T}$ that implements Theorem 1 so that each of the n *subproblems* ψ_i is composed of interval constraints on the input ($\mathcal{X}$), output ($\mathcal{Y}$), and a half-space polytope constraint $Ay \leq b$, for some A, b computed by $\mathcal{T}$. By leveraging $\mathcal{T}$, we achieve a canonical form as each $\psi_i := x \in \mathcal{X}' \wedge y \in \mathcal{Y}' \wedge Ay \leq b$, for a computed $\mathcal{X}', \mathcal{Y}', A, b$. When solving subproblems,

Bounders. `Goose` computes a bound (i.e., a real-valued interval) on every single neuron in C. These bounds are required for several algorithms. In practice, bounding methods are either complete (i.e., exact) or incomplete (i.e., not exact, relaxed, over-approximating). `Goose` includes an incomplete and a complete bounding method. The canonical form bounds input and output variables.

For example, consider the $y = \text{ReLU}(x) = \max(0, x)$ activation function. If y can be bounded by $\ell \leq y \leq u$, then this can be expressed in MILP [177] or SMT (`QF_LIA`) as:

$$(y \leq x - \ell(1 - a)) \wedge (y \geq x) \wedge (y \leq u \cdot a) \wedge (y \geq 0) \wedge a \in \{0, 1\}$$

To compute a complete bounding, `Goose` implements a method based on interval bound propagation [72], which has also been considered in other solvers. Specifically, `Goose` computes a forward interval pass on the input bounds and intersects the resulting output variables with the subproblems' bounds. As each subproblem has different bounds on the input/output variables, this needs to be repeated for each subproblem. The neuron bounds are then updated with a backward pass and computing an intersection from the previous forward pass. This process is repeated until convergence or a fixed number of iterations.

To compute an incomplete bounding, `Goose` leverages the `auto_LiRPA` tool. Incomplete boundings are particularly useful when dealing with activations other than ReLU, as they enable eager translation. Incomplete boundings are computed based on abstraction refinement [162]. During a discussion of the VNN-LIB meeting in 2021, the community with the authors of both tools agreed that DeepPoly and `auto_LiRPA` were equivalent [10].

4.3.3 Prediction Engine and ML-Driven Meta-Solving

Problem Encoder.

One of the biggest challenges in leveraging machine learning for logic solvers lies in designing appropriate features for the problem at hand. `Goose` implements a problem encoder $\xi(C; \psi_i)$ to compute *feature vectors* – a real-valued vector representation of C, ψ_i, with 196 dimensions (features). A complete description of the feature vector used in `Goose` can be found in Tables 4.1, 4.2 in the Appendix.

ML Core.

`Goose` uses an empirical model E based on supervised learning for predicting runtimes and satisfiability. The predictions made by this model are used in both algorithm selection and

probabilistic satisfiability inference. The ML pipeline is generic by implementation, but `Goose` leverages PyTorch [126] with an 8-layer fully-connected ReLU network in its default setting. However, other common platforms such as scikit-learn [130] or XGBoost [49] can also be used.

The empirical models of `Goose` are trained offline. `Goose` has an interface to retrain models and collect data. Retraining models with benchmarks from specific applications can potentially improve performance significantly in practice. However, out of the box, we train on a synthetic dataset [154], which is different from the benchmarks considered in the evaluation later in the chapter (Section 4.4).

The Training Data.

The supervised learning problem solved by `Goose` is a tabular data problem. `Goose` creates feature vectors leveraging the problem encoder ξ, extended with a one-hot encoding of all solvers under consideration. These features are scaled to zero mean and unit variance[5]. The empirical model fits four output target columns. First, the log `PAR-2` score of the benchmark on the solver. The second, third, and fourth columns are to classify `SAT`, `UNSAT`, or `UNKNOWN`. Explicitly, the second, third, and fourth columns are one if and only if the solver benchmark pair returned `SAT`, `UNSAT`, or `UNKNOWN` respectively.

Algorithm Selection at Inference Time.

`Goose` implements algorithm selection by leveraging the predicted `PAR-2` scores (first output column of E). Specifically, for a time budget t, `Goose` constructs a sequential portfolio of solvers and allocates resources based on the predictions of E. First, we create feature vectors for every solver under the problem at hand. Second, we feed these feature vectors through E and obtain the predicted `PAR-2` scores. These predictions can often be far from ground truth and very noisy. However, their relative distances can be very indicative. To exploit this and allocate a time budget, we leverage a `softmin` – a common output activation from deep learning function that computes "smooth" minimum on the predicted log `PAR-2` scores.

[5] $\frac{x-\mu}{\sigma}$, `sklearn.preprocessing.StandardScaler`

Probabilistic Satisfiability Inference.

The canonical form solved by `Goose` is a sequence of subproblems ψ_i over disjunction. If any subproblem ψ_i is `SAT`, then the result of the benchmark is `SAT`. The subsolvers of `Goose` are mostly parallelized. We opt to solve each subproblem ψ_i sequentially. We introduce *probabilistic satisfiability inference* as a way to order these subproblems for solving.

We infer probabilities of satisfiability by leveraging the empirical model E. We compute probabilities by analyzing the second, third, and fourth output columns of E. Similar to algorithm selection, the predicted probabilities are very noisy and must be smoothed. Consider the three-dimensional vector q formed by these three columns. We leverage a `softmax` – a common output activation from deep learning function that computes "smooth" maximum of the predicted log `PAR-2` scores.

Specifically, we create an ordered ranking $\mathcal{P}$ of the subproblems ψ_i on the computation graph C. We compute $\mathcal{P}$ by encoding every subproblem and solver and feeding them through the empirical model E. We compute the `softmax` as described and inspect the `SAT` component to obtain a probability of satisfiability. We calculate an average across all solvers for each subproblem to compute the final ranking $\mathcal{P}$.

Time Iterative Deepening at Inference Time.

We previously defined a *meta-solver* as a solver containing a set of subsolvers that get adaptively called based on information collected online and offline. We next describe *time iterative deepening*, which works in tandem with the algorithm selection and probabilistic satisfiability inference. Specifically, the empirical model E leverages the information collected offline during training while collecting online data by repeatedly building a portfolio and calling subsolvers with the benchmark at hand.

In the beginning, `Goose` allocates a time budget of $t = t_{init}$ seconds. Out of the box, t_{init} is 30 seconds. We first distribute the time t across subproblems by computing a `softmax` on the probabilities of the subproblems obtained from the probabilistic satisfiability inference phase. At this point, we get a time allocation for each subproblem t_{ψ_i}. We now divide the time t_{ψ_i} into a sequential portfolio of all solvers leveraging algorithm selection.

The problem encoder ξ includes several online probing feature vectors. The features of `Goose` were engineered such that when a solver fails on a subproblem in the first iteration of the portfolio, the feature vector changes. When the feature vector changes, the rankings in the algorithm selection phase or the probabilistic satisfiability inference phase can change significantly when constructing portfolios.

For example, one incomplete algorithm that is notably successful on SAT benchmarks is gradient attacking algorithms, such as projected gradient descent (PGD). However, PGD is incomplete and is not suitable for all benchmarks. Suppose Goose is presented with a benchmark, wherein, on the first iteration, the encoding ξ and empirical model E are indicating SAT. The empirical model E may be incorrect (e.g., the benchmark differs significantly from which Goose was trained). In such a scenario, PGD would erroneously allocate significant resources on the first iteration. On subsequent iterations, the encoding of the problem (feature vector) changes, indicating it was more difficult for PGD than expected on the first pass. This allows Goose to correct its original mistake and select a more suitable algorithm with the updated encoding leveraging the information obtained on the first pass. As the problem encoding changes, an entirely new portfolio $\mathcal{S}$ and subproblem rankings $\mathcal{P}$ are generated in future iterations.

4.3.4 The Subsolvers of Goose

Goose implements several decision procedures and semi-decision procedures and configurations thereof. A widely used technique, implemented in Goose, is a direct or eager translation to a core solver, such as MILP. Goose further supports two other complete algorithms, namely, Reluplex and nnenum. This is done by leveraging Marabou's and nnenum's Python API. Goose supports semi-decision procedures Projected Gradient Descent (PGD), random fuzzing, and over-approximation-based bounding leveraging an eager translation.

Eager Blasting.[6] One of the more sophisticated modules of Goose is its eager blasting engine. The engine was designed to be agnostic to the underlying core solver. Goose implements a base class that interacts with the engine. Additional solvers can be added by creating derived classes. The derived class requires methods for syntax tree construction for the operations supported. We provide examples and support for the SCIP [25] and Gurobi [77] MILP solvers in addition to the cvc5 [16] and z3 solver [112] SMT solver (QF_LIRA). Eager blasting requires every neuron to have bounds obtained in the preprocessing phase of Goose. Goose has two bounding algorithms and four back-end solvers for eager blasting, resulting in eight solvers, four complete and four incomplete.

Gradient Methods and Fuzzing. Gradient attacking methods such as projected gradient descent (PGD) have been very successful in discovering adversarial examples. Algorithms for finding adversarial examples and local robustness queries are very active areas

[6] We borrow this term from the SMT community

of research. The canonical form used in `Goose` has an identical syntactic structure, and any such adversarial robustness algorithm can be integrated into `Goose`. Gradient methods can be relatively fast at determining `SAT` results and are bundled in almost every tool in VNN-COMP '21. The competition winner α, β-CROWN implements a highly efficient PGD solver. `Goose` implements two versions of PGD, inspired by the implementation of α, β-CROWN. The difference between the two subsolvers is based on the restart mechanic of the algorithm.

Another technique we implement is fuzzing. Since the canonical form bounds the input variables to an interval, it is straightforward to build an incomplete solver, such as a random fuzzer. The random fuzzer generates inputs uniformly at random and feeds them through C. `Goose` includes a random fuzzer in addition to an extended genetic variant of the fuzzing algorithm.

API Level Solvers. Several other solvers popular within the community have convenient Python APIs that are easily integrated into `Goose`. We use solvers that have fairly orthogonal algorithms to the ones mentioned previously. Specifically, Reluplex – a lazy unwinding of ReLU activations within the simplex algorithm [90] and nnenum – a bound over-approximation algorithm on starsets [9]. Further API solvers can be added swiftly by extending a base solver class.

4.3.5 Algorithmic Description

We next describe Algorithm 1, the main execution loop of `Goose`. For input C, ψ, `Goose` implements the transformation $\mathcal{T}$ corresponding to Theorem 1, specifically, $\mathcal{T}(C, \psi)$ computes the n subproblems $\mathcal{P}$ (line 3). Additionally, a flag $\mathcal{F}$ is used to denote whether or not a subproblem is solved (line 4). The termination condition is the conjunction over $\mathcal{F}$ (i.e., whether or not all problems in $\mathcal{P}$ are `UNSAT`) or if there exists a `SAT` subproblem.

In the beginning of the main loop, `Goose` invokes the prediction engine, first with the problem encoder ξ to create feature vectors[7]. Upon computing all feature vectors, we invoke probabilistic satisfiability inference to rank and create time allocation α_{ψ_i} for every subproblem over the iteration timeout t in time iterative deepening (line 7-8). Next, algorithm selection is invoked for every subproblem to create and allocate a solver portfolio to solve the subproblem. The time to solve the subproblem α_{ψ_i} is divided up across the

[7]In the worst case, an exponential number of subproblems and feature vectors need to be computed, however, this is rarely observed in practice.

Algorithm 2 Main Execution loop of `Goose`

Input: A computation graph C and a linear specification ψ over C
Output: `SAT`/`UNSAT`

1: **procedure** `Goose`-MainLoop
2: $t = t_{init}$
3: $\mathcal{P} = \mathcal{T}(C, \psi)$ ▷ $\mathcal{T}$ is the transform from Theorem 1
4: $\mathcal{F} = [\bot \; \forall C, \psi_i \; \in \mathcal{P}]$ ▷ subproblem flag
5: solved $= \bot$
6: **while** not solved **do**
7: α_{ψ_i} = allocation of t for each unsolved ψ_i from probabilistic satisfiability inference.
8: Sort $\mathcal{P}$ in descending order by α_{ψ_i}
9: **for** C, ψ_i in $\mathcal{P}$ **do**
10: **if** $\mathcal{F}[C, \psi_i]$ **then**
11: **continue**
12: **end if**
13: β_{s,ψ_i} = allocation of α_{ψ_i} for each $s \in \mathcal{S}$ from algorithm selection.
14: Sort $\mathcal{S}$ in descending order by β_{s,ψ_i}
15: **for** $s \in \mathcal{S}$ **do**
16: $\rho = \texttt{run}(s, C, \psi_i, \beta_{s,\psi_i})$
17: **if** ρ is `SAT` **then** ▷ If any subproblem ψ_i is `SAT`
18: **return** SAT
19: **else if** ρ is `UNSAT` **then**
20: $\mathcal{F}[C, \psi_i] = \top$
21: **end if**
22: **end for**
23: **end for**
24: t += an exponential increment
25: solved $= \bigwedge_{v \in \mathcal{F}} v$
26: **end while**
27: **return** `UNSAT`
28: **end procedure**

subsolvers $s \in \mathcal{S}$ as β_{s,ψ_i}. $\mathcal{S}$ is sorted in descending order by β_{s,ψ_i} to use the most likely solver to solve the problem first (lines 13-14).

Now that every subproblem ψ_i has a time allocation of α_{ϕ_i} over the iteration timeout

t and each subsolver s has a time allocation of β_{s,ψ_i} over subproblem timeout α_{ϕ_i} we run the solvers and save the result ρ (line 16). On a successful run (i.e., $\rho \in \{\texttt{SAT}, \texttt{UNSAT}\}$), we either terminate on `SAT` (line 18) or on `UNSAT` mark the problems flag $\mathcal{F}$ as solved (line 20).

At the end of the loop iteration, if the problem remains unsolved, we exponentially increase the time iterative deepening timeout (line 24). On subsequent iterations, the prediction engine updates the encodings and collects new predictions in the probabilistic satisfiability inference and algorithm selection.

4.3.6 Implementation Details, Usage, and Extending `Goose`

Implementation Details.

`Goose` is built on Python 3.8 and consists of nearly 7,000 lines of code. Preprocessing is built leveraging the `onnx` Python packages. The prediction engine has been built with the assistance of `pandas`, `pytorch`, and `scikit-learn`. We implement our own complete forward and backward interval propagation and leverage `auto_LiRPA` for incomplete bounding verification and handling of select activations. d

Usage.

`Goose` can be used as either a command line interface or Python API. When calling `Goose`, a user must specify a path to a computation graph (`.onnx`) and a specification `.vnnlib`. `Goose` has a data collection flag (`-data`) such that when enabled, instead of trying to verify the problem at hand, it runs the subsolvers and collect data from the derived input. The data collected is appended to `Goose`'s offline database. To train the empirical model in `Goose`, the user can run the tool with the training flag (`-train`) for a single epoch.

Extending `Goose`.

The machine learning core, particularly the supervised learning model, has a flexible interface. A compatible model must be able to perform incremental training and multi-output regression. New subsolvers can be added to `Goose` by extending a base class.

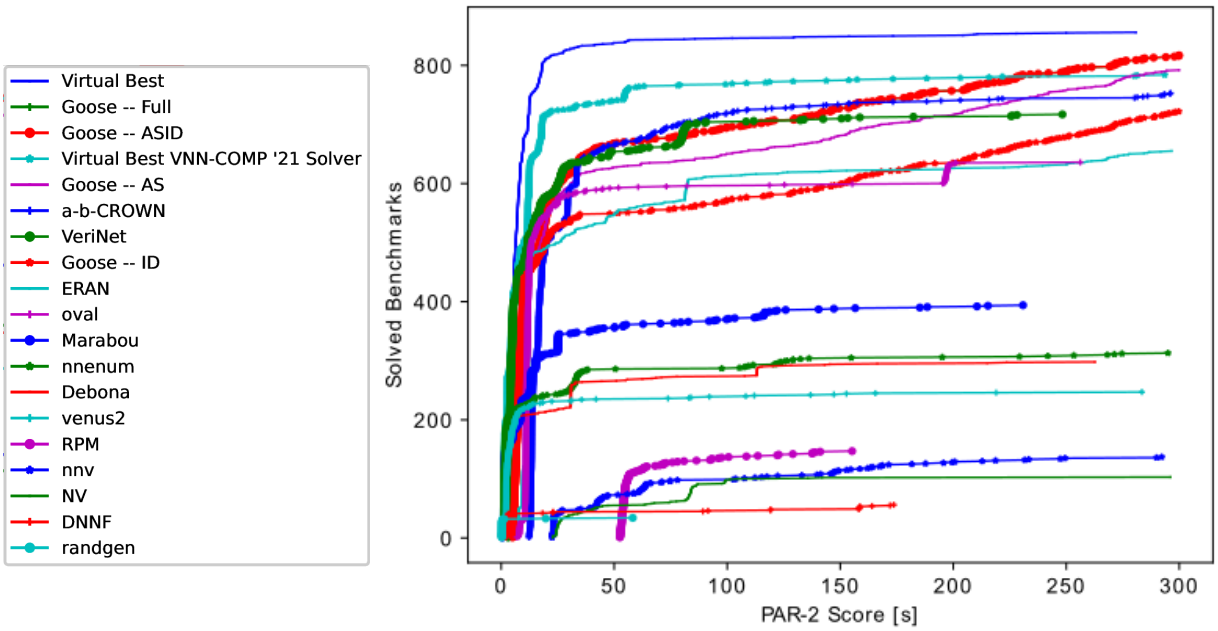


Figure 4.2: Main experimental CDF plot (with ablation study) over VNN-COMP '21 benchmarks (Section 4.4). A CDF is a visualization of a solver's performance on a benchmark suite. The vertical axis represents the number of benchmarks solved (higher is better), and the horizontal axis is the benchmark wise `PAR-2` (lower is better). Further see cumulative `PAR-2` in Table 4.3

4.4 Evaluation on VNN-COMP '21 and '22

In this section, we present an empirical evalutation over VNN-COMP data.

4.4.1 Experimental Setup

Empirical Model Architecture.

`Goose`'s empirical model is built using PyTorch. The empirical model is an 8-layer fully-connected neural network with 128 neurons per layer. Each hidden layer is composed of a linear layer with bias, batch normalization, dropout (p=0.25), and ReLU activation.

Training Setup, Environment, and Data Collection. We generate a training dataset of 10,000 instances produced via a random fuzzer over VNN-LIB [154]. An instance is

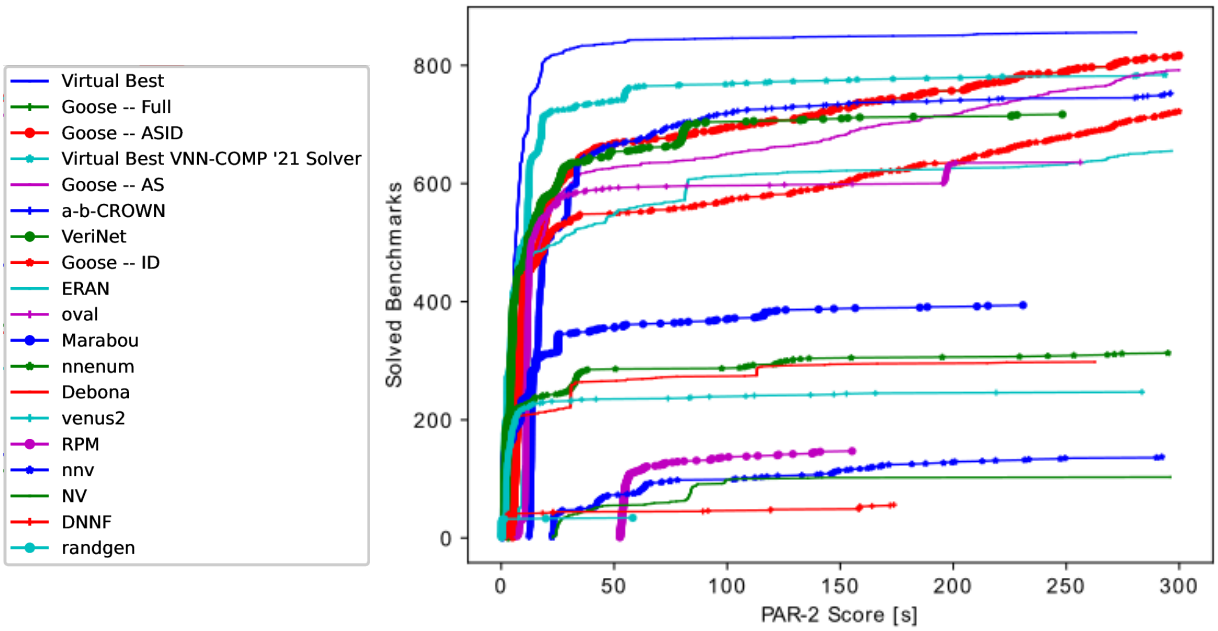


Figure 4.3: Main experimental CDF plot (with ablation study) over VNN-COMP '22 benchmarks (Section 4.4). A CDF is a visualization of a solver's performance on a benchmark suite. The vertical axis represents the number of benchmarks solved (higher is better), and the horizontal axis is the benchmark wise `PAR-2` (lower is better). Further see cumulative `PAR-2` in Table 4.4

generated and solved across all considered solvers, with a 30-second timeout[8]. This data collection is performed on Compute Canada [11], particularly on a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory. Wall-clock runtimes are rounded to the nearest second. All solvers were ran sequentially. For labels we use log `PAR-2` scores. A `PAR-2` is the wall-clock runtime if successful, else twice the wall-clock timeout. The empirical model was trained with Adam [92] and a learning rate of $4 \cdot 10^{-4}$, mean squared error loss, and $1 \cdot 10^{-4}$ weight decay on a NVIDIA 1080 GPU for 2 hours.

[8]This is not used in the experiment and is low because it enables us to increase data collection times

Solver	PAR-2 Score
Virtual Best (Overall)	42764.3
Goose – Full	50764.7
Goose – ASID	50806.7
Virtual Best (VNN-COMP '21 Solver)	52043.5
Goose – AS	67260.2
α, β-CROWN	81455.5
VeriNet	96584.1
Goose – ID	109041.1
ERAN	143568.9
oval	148884.0
Marabou	285863.3
nnenum	333131.4
Debona	342215.7
venus2	364639.4
RPM	435868.4
nnv	440426.1
NV	458472.9
DNNF	483079.8
randgen	494490.2

Table 4.3: Table of sums of `PAR-2` scores across the solvers from the empirical evaluation (Section 4.4). The `PAR-2` score of a solver on a benchmark is the wallclock runtime if successful, otherwise twice the wallclock runtime (lower is better).

Evaluation Environment.

The evaluation was conducted on the Amazon Web Service (AWS), as was the case in VNN-COMP '21. Per VNN-COMP '21 rules, since some tools are either CPU or GPU based, there are two different types of AWS instances depending on the solver [10]. We evaluate `Goose` on the GPU instance (`p3.2xlarge`).

Ablation Study.

We perform an ablation study over `Goose` and its three synergistic strategies. '`Goose`-Full' denotes the full version of the tool. '`Goose`-ASID' denotes the version with algorithm

Solver	PAR-2 Score
Virtual Best	58183.7
Virtual Best VNN-COMP '22 Solver	69764.0
Goose – Full	122882.6
Goose – ASID	138059.9
Goose – AS	147064.7
α, β-CROWN	162076.4
Goose – ID	176078.8
mn_bab	282266.8
verinet	345663.5
nnenum	433870.7
cgdtest	500288.0
marabou	593142.8
peregrinn	614176.0
debona	692136.6
verapak	814516.0
averinn	836569.1
fastbatllnn	877242.2

Table 4.4: Table of sums of `PAR-2` scores across the solvers from the empirical evaluation (Section 4.4). The `PAR-2` score of a solver on a benchmark is the wallclock runtime if successful, otherwise twice the wallclock runtime (lower is better).

selection and iterative deepening, but not probabilistic satisfiability inference. '`Goose`-AS' denotes the version with algorithm selection but not iterative deepening, nor probabilistic satisfiability inference. '`Goose`-ID' denotes the version with iterative deepening but not algorithm selection nor probabilistic satisfiability inference.

Baselines.

We compare `Goose` to all solvers from the VNN-COMP '21 competition. Notably, α, β-CROWN– the winner of the competition overall as well as `nnemum` – the winner of the ACAS Xu benchmarks. Furthermore, we compare against two forms of *virtual best solver*. The virtual best solver denotes the best solver on every single benchmark (including `Goose`). We further consider a virtual best solver just over competition solvers (excluding `Goose`).

4.4.2 Results on VNN-COMP '21 Benchmarks

The simplified (without ablation) CDF plot of the experiment is presented in Figure 4.2. A CDF plot is a visualization of a solver's performance on an instance suite. The vertical axis represents the number of instances solved (higher is better), and the horizontal axis is the instance-wise `PAR-2` (lower is better). In Figure 4.2, we observe that `Goose` outperforms the competition-winning solver α, β-CROWN as well as all other standalone solvers. Table 4.3 (appendix) presents the `PAR-2` across all instances for all solvers. We observe `Goose` to improve on the competition winning solver α, β-CROWN by 37.7% in `PAR-2` score.

4.4.3 Results on VNN-COMP '22 Benchmarks

The simplified (without ablation) CDF plot of the experiment is presented in Figure 4.3. A CDF plot is a visualization of a solver's performance on an instance suite. The vertical axis represents the number of instances solved (higher is better), and the horizontal axis is the instance-wise `PAR-2` (lower is better). In Figure 4.3, we observe that `Goose` outperforms the competition-winning solver α, β-CROWN as well as all other standalone solvers. Table 4.4 (appendix) presents the `PAR-2` across all instances for all solvers. We observe `Goose` to improve on the competition winning solver α, β-CROWN by 25.6% in `PAR-2` score.

4.4.4 Analysis of Results

The incomplete solvers were particularly effective on the first round of the incremental deepening. PGD was extremely effective on `SAT` instances. Complete verifiers have a strong preference for Gurobi. While Gurobi did outperform SCIP in general, it is worth noting that SCIP was up to 7x faster than Gurobi on several instances. One shortcoming of this is that only two of the benchmarks did not contain local robustness queries. Local robustness queries are already in the canonical form and no further subproblems are generated. This limits our ability to empirically evaluate probabilistic satisfiability inference. In the ablation study, '`Goose`-Full' and '`Goose`-ASID' behave identically except on these benchmarks.

Limitations. While `Goose` overcomes several limitations of MachSMT, specifically its ability to leverage online information, common challenges remain. Notably, there are issues in problem encodings, as the encodings of the specification and computation graph lose substantial information.

Moreover, although `Goose` is trained on a fixed dataset, it may encounter difficulties when solving problems that differ significantly from its training data. This issue could be mitigated with future adaptations, unlike MachSMT.

Neural network verifiers like `Goose` struggle with very large inputs. While tools such as `Goose` can be effective on smaller networks, they fail to scale to the size of modern large language models (LLMs).

4.5 Related Work

The biggest motivation for `Goose` are the algorithm selection tools SATZilla [201] and MachSMT [151]. Further, we rely the DNNV project [160] for various useful tools and techniques, such as their neural network simplifier, and make use of their Theorem 1 for our canonical form.

Algorithm and Sequential Portfolio Selection. The field of algorithm selection tools has a rich history and has been around since at least 1976 when Rice et al. first proposed it [138]. Algorithm selectors have been extensively used in many contexts, e.g., classifiers for machine learning [4], combinatorics [95], and other NP-hard optimization problems [175, 181]. Within the context of solvers, algorithm selectors have been proposed for SAT [196, 200, 201], Quantified Boolean Formulas [104, 133], and for SMT Solvers [151]. A system for dynamic algorithm selection over SMT solvers was also recently proposed [132]. `Goose` differs from these previous approaches in that our algorithm selection is adapative and is in turn is used to construct adaptive sequential portfolios.

Probabilistic Satisfiability Inference and ML for Solvers. NeuroSAT is a SAT solver that learns a classifier via single-bit supervision [159] and NeuroCore is a simplified variant that predicts variable activity based on the likelihood of appearing in the UNSAT core [158]. MapleSAT is a SAT solver that leverages reinforcement learning for branching [102]. Training models to predict SAT was studied by SATZilla [200].

Time Iterative Deepening. The concept of time iterative deepening has been explored in the context of heuristic/task switching for SAT [35] and the DASH method for MILP solvers [103]. In the context of `Goose`, we leverage time iterative deepening to collect performance histories of solvers on a given input, in an online fashion in one iteration of `Goose` algorithm. We use that information to adaptively modify the algorithm selection for the subsequent iteration.

4.6 Conclusions

In this chapter, we present `Goose`, a meta-solver for NN verification. By meta-solver, we mean a tool containing a set of subsolvers that get adaptively called, in a sequence, based on online and offline information collected about their performance histories on a given input. `Goose` has a meta-solver architecture (Figure 4.1) and supports a wide variety of incomplete and complete solvers (some of which were implemented by us). `Goose` leverages three key meta-solving techniques, namely, adaptive algorithm selection, probabilistic satisfiability inference, and time interval deepening to implement an adaptive sequential portfolio of solvers for NN verification.

We evaluate the efficacy of our `Goose` meta-solver on the VNN-COMP 2021 (resp. VNN-COMP 2022) benchmarks against 13 (resp. 11) state-of-the-art neural network verification solvers. We observed a 37.7% (resp. 25.6%) improvement across benchmarks and solvers from VNN-COMP '21 (resp. VNN-COMP '22).

The success of `Goose` for the NN verification problem suggests that similar meta-solving techniques (i.e., adaptive sequential portfolio methods) can effective for the SAT/SMT problem as well. Hence, in the future, we plan to extend our techniques for constructing meta-solvers over SAT and SMT solvers.

Chapter 5

Reinforcement Learning based Performance Fuzzing of SMT Solvers

This chapter presents BanditFuzz, a multi-agent reinforcement learning (RL) guided performance fuzzer for state-of-the-art Satisfiability Modulo Theories (SMT) solvers. BanditFuzz constructs inputs that expose performance issues in a set of target solvers relative to a set of reference solvers, and is the first performance fuzzer that supports the entirety of the theories in the SMT-LIB (v2.6) initiative. Another useful feature of BanditFuzz is that users can specify the size of inputs they want, thus enabling developers to construct very small inputs that zero-in on a performance problem in their SMT solver relative to other competitive solvers. We evaluate BanditFuzz across 52 logics from SMT-COMP '20 targeting competition-winning solvers against runner-ups. We baseline BanditFuzz against random fuzzing and a single agent algorithm and observe a significant improvement, with up to a 82.6% improvement in the margin of `PAR-2` scores across baselines on their respective benchmarks. Furthermore, we reached out to developers and contributors of the CVC4, Z3, and Bitwuzla solvers and provide case studies of how BanditFuzz was able to expose surprising performance deficiencies in each of these tools.

5.1 Motivation

Performance Fuzzing for Satisfiability Modulo Theories. In recent years, efficient Satisfiability Modulo Theories (SMT) solvers have dramatically impacted many areas of software engineering and security. Applications of these tools range from program analy-

sis [46, 88, 74], synthesis [168, 136] , model checking [93, 55, 6], test case generation [44], and neural network verification [89], to name just a few.

With efficient SMT solvers being the catalyst for numerous developments in academia and industry, there is an insatiable demand for evermore powerful solvers. To this end, researchers have spent decades optimizing these tools. Regrettably, despite these advances, SMT solvers are prone to hard-to-find performance deficiencies. While the worst-case complexity of the problems solved by SMT solvers can be very high, they can be frustratingly slow on relatively simple formulas. Such performance deficiencies can be due to developer oversight (e.g., missing rewrite rules or unoptimized code and data structures) or the result of hard-to-entangle interactions of solver heuristics. If solvers are to continue to impact industry and fuel further research, it is imperative that there be an initiative to find and eliminate such performance deficiencies where possible.

In this chapter, we make a case for the use of software performance fuzzing [173, 171, 105] to systematically find such deficiencies in state-of-the-art SMT solvers. Software fuzzing techniques have had tremendous impacts in making SMT Solvers more robust [31, 41, 189, 121], and there is no reason why performance fuzzers cannot have a similar impact. While it is still a relatively a new field, performance fuzzing is already showing promise in many domains despite the difficulty of the problem of finding suitable inputs that expose performance issues in programs-under-test [100, 87].

This chapter presents the BanditFuzz tool, a performance fuzzer that supports the entirety of the theories in SMT-LIB. We define the notion of "performance issue", in the SMT solver setting, in a relative sense. That is, we say that solver A is less performant on an input I relative to solver B, if solver B (that supports the same input language as A) can solve I significantly faster. This is very natural, since if both solvers-under-test are not able to solve an input, it doesn't unambiguously point to a performance issue in either. However, when one solver is significantly faster than a competing one on a given input, there is no question that the slower solver has a performance issue.

How BanditFuzz works: The input to BanditFuzz is a set of target solvers, a set of reference solvers, and a constraint (e.g., size of input desired, the input language of the solvers), and its output is a single benchmark or test input, such that a quantity we refer to as the "performance margin" between the target solver and reference solver is maximized (the tool is designed to be run over multiple processes to create a benchmark suite). Intuitively, the performance margin can be defined as difference between the runtimes (or `PAR-2` scores [109]) of a target and a reference solver on a given input.

Internally, BanditFuzz uses a two-agent reinforcement learning (RL) method to mutate a randomly-generated input such that over time the performance margin between a

target and a reference solver is maximized. More precisely, first an input benchmark is randomly-generated and queried across all solvers. One of the agents learns how to mutate a benchmark by inserting and replacing the grammatical constructs of the SMT-LIB language, respecting the size constraints set forth by the user. More precisely, this agent manages an exploration vs. exploitation trade-off between trying new grammatical constructs (explore) vs. inserting ones that have been shown to increase the performance margin (exploit). The other agent manages the exploration vs. exploitation trade-off between generating new inputs (explore) or mutating the best-observed input (exploit). Fuzzers, including single-agent RL fuzzers, are notorious for getting stuck in local minima [144, 58, 106, 67]. This two-agent RL method, by contrast, may avoid getting stuck in local minima.

5.2 BanditFuzz

In this section, we describe our technique, BanditFuzz, a grammar-based mutation fuzzer that uses reinforcement learning (RL) to efficiently isolate grammatical constructs of an input that are the cause of a performance issue in a solver-under-test. The ability of BanditFuzz to isolate those grammatical constructs that trigger performance issues, in a blackbox manner, is its most interesting feature. The architecture of BanditFuzz is presented in Figure 5.1.

5.2.1 Description of the BanditFuzz Algorithm

BanditFuzz takes as input a grammar G that describes well-formed inputs to a set P of solvers-under-test (for simplicity, assume P contains only two programs, a target program T to be fuzzed, and a reference program R against which the performance or correctness of T is compared), a fuzzing objective (e.g., aim to maximize the relative performance difference between target and reference solvers) and outputs a ranked list of grammatical constructs (e.g., syntactic tokens or keywords over G) in the descending order of ones that are most likely to cause performance issues. We infer this ranked list by extrapolating from the policy of the RL agent. It is assumed that BanditFuzz has blackbox access to the set P of the solvers-under-test.

The BanditFuzz algorithm works as follows: BanditFuzz generates well-formed inputs that adhere to G and mutates them in a grammar-preserving manner (the instance generator and mutator together are referred to as fuzzer in Figure 5.1) and deploys an RL agent

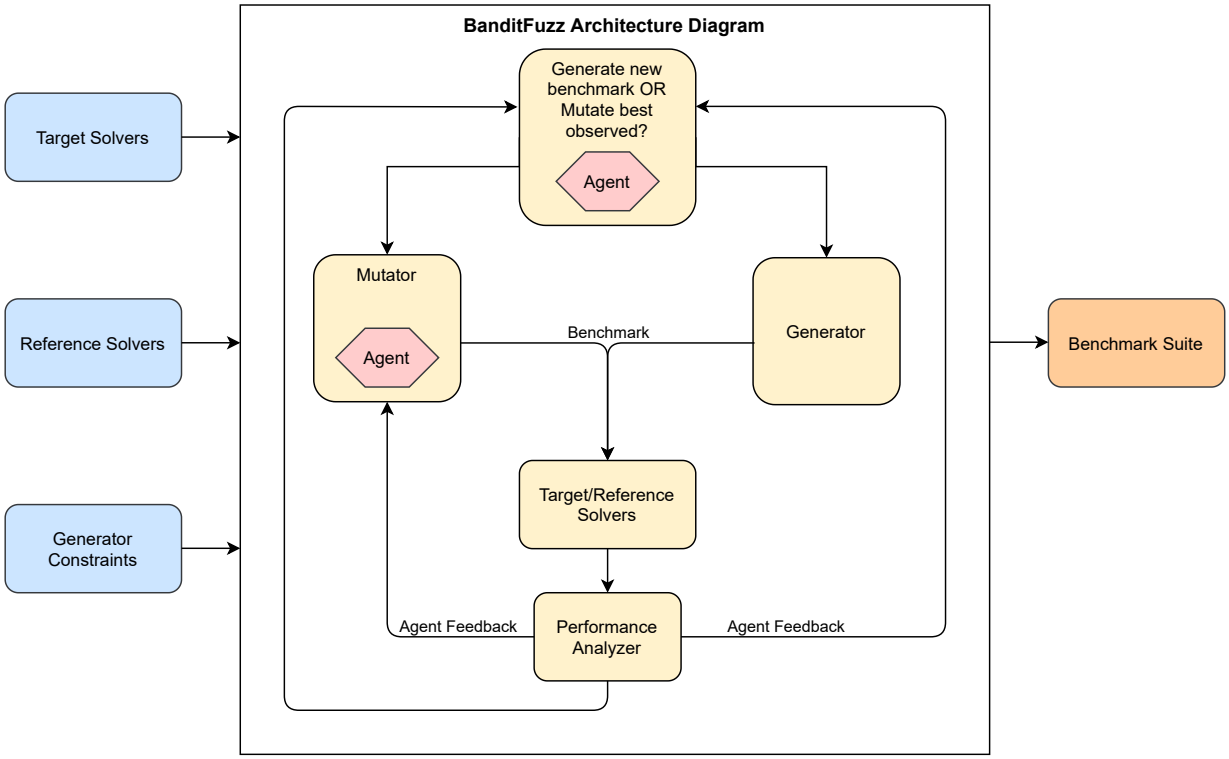


Figure 5.1: **Architecture Diagram of BanditFuzz**. The BanditFuzz tool deploys two unique agents: one is a mutator agent that learns how to mutate the best observed input, while the other agent aims to assist in the prevention of getting stuck in local minima. Both agents learn an action selection policy in a feedback loop based on the empirically collected data over the course of running the target and reference solvers over the generated benchmarks.

(specifically a MAB agent) within a feedback loop to learn which grammatical constructs of G are the most likely culprits that cause performance issues in the target program T in P.

BanditFuzz reduces the problem of how to mutate an input to an instance of the MAB problem. As discussed earlier, in the MAB setting an agent is designed to maximize its cumulative rewards by selecting the arms (actions) that give it the highest expected reward, while maintaining an exploration-exploitation tradeoff. In BanditFuzz, the agent chooses actions (grammatical constructs used by the fuzzer to mutate an input) that maximize the reward over a period of time (e.g., increasing the runtime difference between the target solver T and a reference solver R). It is important to note that the agent learns an action

selection policy via a historical analysis of the results of its actions over time. Within its iterative feedback loop (that enables rewards from the analysis of solver outputs to the RL agent), BanditFuzz observes and analyzes the effects of the actions it takes on the solvers-under-test. BanditFuzz maintains a record of these effects over many iterations, analyzes the historical data thus collected, and zeroes-in on those grammatical constructs that have the highest likelihood of reward. At the end of its run, BanditFuzz outputs a ranked list of grammatical constructs which are most likely to cause performance issues, in descending order. In the fuzzing for relative performance fuzzing mode, BanditFuzz performs the above-described analysis to produce a ranked list of grammatical constructs that increase the difference in running time between a target solver T and a reference solver R.

5.2.2 Instance Generator and Grammar-preserving Mutator

BanditFuzz's fuzzer (See Architecture of BanditFuzz in Figure 5.1) consists of two sub-components, namely, an instance[1] generator and a grammar-preserving mutator (or simply, mutator). The instance generator is a program that randomly samples the space of inputs described by the grammar G. The mutator is a program that takes as input a well-formed G-instance and a grammatical construct δ and outputs another well-formed G-instance.

Instance Generator: Here we describe the generator component of BanditFuzz, as described in Figure 5.1. Initially, BanditFuzz generates a random well-formed instance using the input grammar G (FP or string SMT-LIB grammar) via a random abstract syntax tree (AST) generation procedure built into StringFuzz [31]. We generalize this procedure for the theory of FP.

The FP input generation procedure works as follows: we first populate a list of free 64-bit FP variables and then generate random ASTs that are asserted in the instance. Each AST is rooted by an FP predicate whose children are FP operators chosen at random. We deploy a recursive process to fill out the tree until a predetermined depth limit is reached. Leaf nodes of the AST are filled in by randomly selecting a free variable or special constant. Rounding modes are filled in when required by an operator's signature. The number of variables and assertions are parameters to the generator and are specified for each experiment.

Similar to the generator in StringFuzz, BanditFuzz's generation process is highly configurable. The user can choose the number of free variables, the number of assertions, the

[1]We use the terms "instance" and "input" interchangeably through this chapter.

maximum depth of the AST, the set of operators, and rounding terms. The user can also set weights for specific constructs as a substitute for the default uniform random selection.

Grammar-preserving Mutator: The second component of the BanditFuzz fuzzer is the mutator. In the context of fuzzing SMT solvers, a mutator takes a well-formed SMT formula I and a grammatical construct δ as input, and outputs a *mutated* well-formed SMT formula I' that is like I, but with a suitable construct (say, γ) replaced by δ. The construct γ in I could be selected using some user-defined policy or chosen uniform-at-random over all possible grammatical constructs in I. In order to be grammar-preserving, the mutator has to choose γ such that no typing and arity constraints are violated in the resultant formula I'. The grammatical construct δ, one of the inputs to the mutator, may be chosen at random or selected using an RL agent. We describe this process in greater detail in the next subsection.

On the selection of a grammatical construct, an arbitrary construct of the same type (predicate, operator, or rounding mode, etc.) is selected uniformly at random. If the replacement involves an arity change, the rightmost subtrees are dropped on a decrease in arity, or new subtrees are generated on the increase in arity.

For illustrative purposes, we provide an example mutation here. Consider a maximum depth of two, fixed set of free FP variables (x_0, x_1), limited rounding mode set of $\{RNE\}$, and an asserted equation:

$$(fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.sub\ RNE\ x_0\ x_1)).$$

If the agent elects to insert $fp.abs$ there are two possible results:

$$(fp.eq\ (fp.abs\ x_0)(fp.sub\ RNE\ x_0\ x_1)), \quad (fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.abs\ x_0)).$$

For further analysis, consider the additional asserted equation:

$$(fp.eq\ (fp.abs\ x_0)(fp.abs\ x_1)),$$

if the agent elects to insert $fp.add$, then there are four[2] possible outputs:

$$(fp.eq\ (fp.add\ RNE\ x_0\ x_0)(fp.abs\ x_1))$$
$$(fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.abs\ x_1))$$
$$(fp.eq\ (fp.abs\ x_0)(fp.add\ RNE\ x_1\ x_0))$$

[2]This is assuming only the RNE rounding mode is allowed, otherwise each of the below expressions could have any valid rounding mode resulting in 20 possible outputs.

Algorithm 3 BanditFuzz's Performance Fuzzing Feedback Loop. Also refer to BanditFuzz architecture in Figure 5.1.

```
1: procedure BanditFuzz(G)
2:     Instance I ← a randomly-generated instance over G                ▷ Fuzzer
3:     Run target solver T and reference solver(s) R on I
4:     Compute PerfScore(I)                                             ▷ OutputAnalyzer
5:     θ = 2· Solver timeout
6:     while fuzzing time limit not reached and PerfScore(I) < θ do
7:         construct ← RL AGENT picks a grammatical construct           ▷ RL Agent
8:         I' ← Mutate I with construct                                 ▷ Fuzzer
9:         Run target solver T and reference solver(s) R on I'
10:        if PerfScore(I', P) > PerfScore(I, P) then                   ▷ OutputAnalyzer
11:            Provide reward to RL AGENT for construct
12:            I ← I'
13:        else
14:            Provide no reward to AGENT for construct
15:        end if
16:    end while
17:    return I and the ranking of constructs from RL AGENT
18: end procedure
```

$$(fp.eq\ (fp.abs\ x_0)(fp.add\ RNE\ x_1\ x_1))$$

In these examples, the reason why the possible outputs may seem limited is due to type and arity preservation rules described above. As described below, the fuzzer would select one of the mutations in the above example in a manner that maximizes expected reward (e.g., the fuzzing objective such that the performance difference between a solver-under-test and a reference solver is increases).

5.2.3 Agents and Reward-driven Feedback Loop in BanditFuzz

As shown in Figure 5.1, the key component of BanditFuzz is an RL agent (based on Thompson sampling) that receives rewards and outputs a ranked list of grammatical constructs (actions). The fuzzer maintains a policy and selects actions from it ("pulling an arm" in the MAB context), and appropriately modifies the current input I to generate a novel input I'. The rewards are computed by the Output Analyzer, which takes as input

the outputs and runtimes produced by the solver-under-test S and computes scores and rewards appropriately. These are fed to the RL agent; the RL agent tracks the history of rewards it obtained for every grammatical construct and refines its ranking over several iterations of BanditFuzz's feedback loop (see Algorithm 3). In the following subsections, we discuss it in detail.

Computing Rewards for Performance Fuzzing: We describe BanditFuzz's reward computation for performance fuzzing in detail here and display the pseudo-code for it in Algorithm 3 (see also the architecture in Figure 5.1 to get a higher-level view of the algorithm). Initially, the fuzzer generates a well-formed input I (sampled uniformly at random). BanditFuzz then executes both the target solver T and reference solver R on I and records their respective runtimes (it is assumed that both solvers may produce the correct answer with respect to input I or timeout). BanditFuzz's Performance Analyzer module then computes a score, to which the reward of the agent is derived.

Formally, BanditFuzz solves a search problem to find a solver input I over the language L that maximizes the performance margin between T and R

$$\max_{I \in L} \phi(T, R, I)$$

where ϕ is a scoring function. In this chapter, we will exclusively consider a scoring function of the `PAR-2` score margin between the best-performing target solver and worst-performing reference solver. More formally, we score each input I with respect to T, R as follows:

$$\phi(T, R, I) = \min_{t \in T}(\texttt{PAR-2}(t, I)) - \max_{r \in R}(\texttt{PAR-2}(r, I))$$

where the `PAR-2` function returns twice the wallclock timeout if the solver fails to solve the input, otherwise, the wallclock runtime. `PAR-2` is a useful metric that quantifies a tools' performance over a benchmark suite and is used to determine winners in the SAT competitions [109]. These calculations correspond to the 'Performance Analyzer' in Figure 5.1.

PerfScore, defined as

$$\text{PerfScore}(I) := \text{runtime}(I, T) - \text{runtime}(I, R)$$

where the quantity $\text{runtime}(I, T)$ refers to the wall clock runtime of the target solver T on I, and $\text{runtime}(I, R)$ the runtime of the reference solver R on I. If the target solver reaches the wallclock timeout, we set $\text{runtime}(I, T)$ to be $2 \cdot timeout$ — PAR-2 scoring in the SAT competition. In the same iteration, BanditFuzz mutates the input I to a well-formed input

I' and computes the quantity PerfScore(I'). Recall that we refer to the mutation inserted into I to obtain I' as γ.

The OutputAnalyzer then computes the rewards as follows. It takes as input I, I', quantities PerfScore(I), and PerfScore(I'), and if the quantity PerfScore(I') is better than PerfScore(I) (i.e., the target solver is slower than the reference solver on I' relative to their performance on I), the mutations γ gets a positive reward, else it gets a negative reward. Recall that we want to reward those constructs which make the target solver slower than the reference one. The reward for all other grammatical constructs remains unchanged.

The rewards thus computed are fed into the RL agent. The bandit then updates the rank of the grammatical constructs. The Thompson sampling bandit analyzes historically, the positive and negative rewards for each grammatical construct and computes the α and β parameters. The highest-ranked construct γ is fed into the fuzzer for the subsequent iteration. This process continues until the fuzzing resource limit has been reached.

5.2.4 Performance Margins and Scoring

Formally, BanditFuzz solves a search problem to find a solver input I over the language L that maximizes the performance margin between T and R

$$\max_{I \in L} \phi(T, R, I)$$

where ϕ is a scoring function. In this chapter, we will exclusively consider a scoring function of the `PAR-2` score margin between the best performing target solver and worst performing reference solver. More formally, we score each input I with respect to T, R as follows:

$$\phi(T, R, I) = \min_{t \in T}(\texttt{PAR-2}(t, I)) - \max_{r \in R}(\texttt{PAR-2}(r, I))$$

where the `PAR-2` function returns twice the wallclock timeout if the solver fails to solve the input, otherwise, the wallclock runtime. `PAR-2` is a useful metric that quantifies a tools' performance over a benchmark suite and is used to determine winners in the SAT competitions [109]. These calculations correspond to the 'Performance Analyzer' in Figure 5.1.

5.2.5 Multi-Agent Fuzzing

This chapter presents an implementation of the BanditFuzz algorithm with three key improvements. First, BanditFuzz supports all theories in the SMT-LIB standard [18] and

all their combined corresponding logics. The action set of the mutator agent is the set of grammatical constructs across all enabled logics.

The second major change is that this tool is now multi-agent. The BanditFuzz tool includes a second agent to assist in preventing the tool from getting stuck in local minima, which is a major problem in fuzzing in general [144, 58, 106, 67]. A key contribution of this chapter is an additional agent with the following **action set**: {Mutate the best observed benchmark, Randomly Sample from L}. The previous approach had a fixed alternation scheme of randomly sampling and mutating the best observed input, which posteriori, resulted in the algorithm frequently getting stuck in local minima, as it is oblivious to all previously collected empirical data. The second agent uses a similar **reward signal** if the most recent benchmark improves on the best observed benchmark, reward is received, otherwise no reward is received.

5.3 Implementation and Engineering

In this section, we discuss some of the implementation and the engineering of BanditFuzz. The BanditFuzz tool is written in Python3 and contains 5,000 lines of code. BanditFuzz is lightweight with minimal dependencies and can be installed in seconds.

Input/Output: Reference/Target solvers are provided to BanditFuzz as paths to an executable file which acts as an interface to the solver. These executable files will run internally within BanditFuzz during its main runtime loop. The generator constraints are command-line arguments bounding formula sizes with several theory specific constraints (e.g., bit-width, UF arity, etc.). The output of BanditFuzz is a directory of benchmarks with timing and memory analysis. A single run BanditFuzz will produce a single benchmark. To build a benchmark suite, BanditFuzz can safely be run in parallel and tested on several major cloud computing environments (e.g., AWS). We provide an interface that allows for this to be done via the command line.

Generator: The generator module is responsible for producing well-formed SMT-LIB inputs. Internally, BanditFuzz an Abstract Syntax Tree (AST) data structure to represent a benchmark. Each AST is asserted with root nodes of a boolean sort and positive arity. Each AST is populated by randomly sampling from the set constructs of the required sort with leaf nodes of variables or theory literals. All ASTs are full with respect to the maximum depth specified by the user.

Mutator: A mutator is a python method that takes as input a benchmark and a grammatical construct. The output is a perturbation of the input benchmark that contains a

novel occurrence of the input construct. The mutator works by constructing the set nodes of the input construct's sort and then uniformly at random replaces the selected construct with the input construct. The mutator then applies a procedure to ensure the resulting benchmark remains well-formed as the node replacement may result in an arity change. This is done by deleting or generating new subtrees that are consistent with the generator constraints. Like the generator, the resulting AST from the mutator is full with respect to the maximum depth.

Agents: We use a context-free MAB approach in this chapter for our agent, namely Thompson Sampling. However, in principle, this can be lifted to several more RL paradigms. Our agent implementation is lightweight and makes a single external API call (i.e., the NumPy beta distribution sampling method [80]). Furthermore, unlike several RL paradigms, our MAB solution only has a single hyper-parameter, the exponential decay of the observed empirical mean.

Performance Analyzer: We include a performance analyzer to monitor the subprocesses' resource consumption (e.g., wall-clock runtimes and memory). Processes are killed if they violate the user's constraints. When calling solvers, target solvers are run first under the provided constraints. Afterward, reference solvers are ran using a dynamic timeout scheme based best-observed performance margin. This prevents wasting time on the reference solvers when it is no longer possible for the current benchmark to have a higher margin than the best observed.

5.4 Usage

In this section, we demonstrate how to use BanditFuzz. The BanditFuzz package has two core tools:

- `smtfuzz` – A fuzzer (i.e., a tool that generates inputs to a program-under-test) for all SMT-LIB theories. In principle, this fuzzer can be used in any fuzzing context, but in this chapter it is the core fuzzer in BanditFuzz's fuzzing algorithm.
- `banditfuzz` – An implementation of the BanditFuzz performance fuzzing algorithm. This program calls `smtfuzz` in a loop and inherits all of its command line arguments.

Argument	Description
`--num-asserts`	Set the number of assertions in the generated benchmark
`--depth`	Set the depth of each asserting AST
`--num-vars`	The number of theory variables
`-q --quantifiers`	Enable quantifiers
`-a --arrays`	Enable arrays
`-uf --uninterpreted-functions`	Enable uninterpreted functions
`-str --strings`	Enable strings
`-fp --floating-point`	Enable floating-point
`-bv --bit-vectors`	Enable bit-vectors
`-int --integer`	Enable integers
`-r --real`	Enable reals
`-8 -16 -32 -64 -128 -256`	Bit width for bit-vectors and floating-point arithmetic
`-l --linear`	Enforce integer and real constraints to be linear

Table 5.1: Sample of generator arguments for the BanditFuzz tool

5.4.1 Using `smtfuzz`

The `smtfuzz` tool generates random Abstract Syntax Trees (ASTs) based on the enabled theories. `smtfuzz` is designed to be extremely flexible. Users can modify the problem size easily by setting the `--num-asserts` and `--depth` parameters to increase the number of assertions and size of each assertion respectively. The generator in `smtfuzz` supports all core theories in the SMT-LIB initiative [18]. Each theory can be enabled by setting its respective flag. `smtfuzz` will automatically set the problem's logic based on the enabled theories.

5.4.2 Using `banditfuzz`

The `banditfuzz` script is an implementation of the BanditFuzz algorithm and uses `smtfuzz` as its primary fuzzer. Furthermore, it has four key additional arguments:

1. `--target-solvers` – The set of executables that BanditFuzz will try to expose relative performance deficiencies on.
2. `--reference-solvers` – The set of reference executables that BanditFuzz will try to expose relative performance deficiencies with respect to.

```
$ smtfuzz -qf -bv -fp - uf --num-asserts 1 --num-vars 1 --num-ufs 1
(set-logic QF_UFBVFP)
(declare-const bool_0 Bool)
(declare-const fp_0 (_ FloatingPoint 8 24))
(declare-const bv_0 (_ BitVec 32))
(declare-fun uf_0 (Bool (_ BitVec 32) Bool Bool (_ FloatingPoint 8 24)) Bool)
(assert (uf_0 (fp.isPositive (fp.roundToIntegral RTZ fp_0)) (bvsub (bvsmod bv_0
#x2ad75270 ) (bvxnor bv_0 #x3a990975 )) (fp.isNaN (fp.abs fp_0)) (bvuge (bvor bv_0
bv_0) (bvnor #x0a1b63c9  #x52911167 )) (fp.roundToIntegral RTN (fp.neg (fp #b1
#b11110100 #b11000100101000110101000)))))
(check-sat)
(exit)
```

Figure 5.2: Example usage of `smtfuzz` to generate a benchmark in the logic of QF_UFBVFP

3. `--query-timeout` – This parameter is the wallclock timeout of each query of a solver on an input benchmark.

4. `--global-timeout` – This parameter is the global timeout of `banditfuzz`. When this time is met, `banditfuzz` will return the benchmark that had the highest performance margin between the target solvers and the reference solvers.

5.5 Evaluation on SMT-LIB and Solvers

In this section, we present an evaluation of BanditFuzz vs. standard performance fuzzing algorithms.

5.5.1 Experimental Setup

Experimental Objective: Here we describe our evaluation of BanditFuzz against random fuzzing and previous similar work by Scott et al. [149]. The objective of the experiment

[3]The previous code framework by Scott et al. [149] only supports two logics QF_FP and QF_S. When evaluating outside of these logics, we baseline by using the BanditFuzz code base with the second agent disabled.

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on
			Random	BanditFuzz	Baseline [%]
ABVFP	CVC4	Z3	2,716	5,579	105
ABVFPLRA	CVC4	Z3	21,376	60,000	181
ALIA	CVC4	Z3	4,238	11,340	168
ANIA	CVC4	Z3	34,883	60,000	72
AUFLIA	CVC4	Z3	34,229	60,000	75
AUFLIRA	CVC4	Z3	7,650	30,428	298
AUFNIA	CVC4	Z3	279	753	170
AUFNIRA	CVC4	Z3	7,967	16,949	113
BV	CVC4	Z3	50,561	60,000	19
BVFP	CVC4	Z3	319	758	138
BVFPLRA	CVC4	Z3	50,844	60,000	18
FP	CVC4	Z3	3,700	10,674	188
FPLRA	CVC4	Z3	15,325	49,528	223
LIA	CVC4	Z3	5,635	19,050	238
LRA	CVC4	Z3	11,184	25,560	129
NIA	CVC4	Z3	48,752	60,000	23
NRA	CVC4	Z3	27,066	60,000	122
QF_ABV	Bitwuzla	Yices2	16,280	45,814	181
QF_ABVFP	Bitwuzla	CVC4	48,484	60,000	24
QF_ABVFPLRA	CVC4	COLIBRI	1,652	4,431	168
QF_ALIA	Yices2	Z3	17,670	60,000	240
QF_ANIA	CVC4	Z3	34,444	60,000	74
QF_AUFBV	Yices2	Bitwuzla	5,375	14,704	174
QF_AUFLIA	Yices2	CVC4	21,836	56,345	158
QF_AUFNIA	CVC4	Z3	35,817	60,000	68
QF_AX	Yices2	CVC4	3,251	5,153	59

Table 5.2: Table of results comparing BanditFuzz to Random fuzzing across logics of SMT-COMP '20. The improvement column is the percentage improvement of BanditFuzz over Random Fuzzing. Rows are sorted alphabetically by logic.

is as follows: given the same amount of resources, which of the three tools maximizes the performance margin for a given target solver vs. a set of reference solvers over all the 52 logics used in SMT-COMP '20. For target solvers, we chose the most performant solvers from the SMT-COMP '20 competition, and as reference solvers, we used the runner-up

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on
			Random	BanditFuzz	Baseline [%]
QF_BV	Bitwuzla	CVC4	22,142	52,681	138
QF_BVFP	Bitwuzla	CVC4	29,949	60,000	100
QF_BVFPLRA	CVC4	COLIBRI	23,053	55,228	140
QF_FP	Bitwuzla	COLIBRI	37,692	60,000	59
QF_FPLRA	COLIBRI	CVC4	2,030	4,053	100
QF_LIA	CVC4	Yices2	3,217	5,399	68
QF_LIRA	Yices2	CVC4	1,795	7,584	323
QF_LRA	CVC4	Yices2	4,571	14,184	210
QF_NIA	CVC4	Yices2	32,540	60,000	84
QF_NIRA	CVC4	Yices2	8,348	32,509	289
QF_NRA	Yices2	CVC4	22,861	60,000	162
QF_S	CVC4	Z3str4	35172	60,000	71
QF_SLIA	CVC4	Z3str4	3,956	15,381	289
QF_UF	Yices2	Z3	5,607	15,362	174
QF_UFBV	Yices2	Bitwuzla	34,315	60,000	75
QF_UFFP	Bitwuzla	COLIBRI	12,909	20,373	58
QF_UFLIA	Yices2	CVC4	1,696	2,428	43
QF_UFLRA	Yices2	Z3	8,431	26,529	215
QF_UFNIA	CVC4	Yices2	3,864	13,564	251
QF_UFNRA	Yices2	CVC4	53,374	60,000	12
UF	CVC4	Z3	3,469	13,368	285
UFBV	CVC4	Z3	37,751	60,000	59
UFLIA	CVC4	Z3	174	868	399
UFLRA	CVC4	Z3	1,567	5,159	229
UFNIA	CVC4	Z3	5,671	17,419	207
UFNRA	CVC4	Z3	17,219	60,000	248

Table 5.3: Table of results comparing BanditFuzz to Random fuzzing across logics of SMT-COMP '20. The improvement column is the percentage improvement of BanditFuzz over Random Fuzzing. Rows are sorted alphabetically by logic.

solver(s) from the same track in the competition. In the case where a solver was not able to run in our setup, often due to environmental hard-codings, we replaced it with the next most performant alternative.

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on
			Scott et al. [149]	BanditFuzz	Baseline [%]
QF_FP	Bitwuzla	COLIBRI	51,893	60,000	14.4
QF_S	CVC4	Z3str4	53,231	60,000	11.9
ABVFPLRA	CVC4	Z3	46,237	60,000	25.9
ANIA	CVC4	Z3	54,120	60,000	10.3
AUFLIRA	CVC4	Z3	22,314	30,428	30.7
FP	CVC4	Z3	9,109	10,674	15.8
FPLRA	CVC4	Z3	31,808	49,528	43.5
LIA	CVC4	Z3	17,009	19,050	11.3
LRA	CVC4	Z3	16,098	25,560	45.4
NRA	CVC4	Z3	39,116	60,000	42.1
QF_ALIA	Yices2	Z3	35,912	60,000	50.2
QF_ANIA	CVC4	Z3	56,198	60,000	6.5
QF_AUFBV	Yices2	Bitwuzla	11,103	14,704	27.9
QF_BVFPLRA	CVC4	COLIBRI	47,180	55,228	15.7
QF_LIRA	Yices2	CVC4	3,152	7,584	82.6
QF_NIA	CVC4	Yices2	58,199	60,000	3.0
QF_NIRA	CVC4	Yices2	17,009	32,509	62.6
QF_NRA	Yices2	CVC4	42,188	60,000	34.9
QF_UFLRA	Yices2	Z3	22,092	26,529	18.3
QF_UFNIA	CVC4	Yices2	9,917	13,564	31.1
UFNIA	CVC4	Z3	14,282	17,419	19.8
UFNRA	CVC4	Z3	27,901	60,000	73.0

Table 5.4: Table of select results comparing BanditFuzz to the work of Scott et al. [149] across select logics. The improvement column is the percentage improvement of BanditFuzz over the baseline.[3]

Baselines: As baselines, we used random fuzzing (i.e., `smtfuzz` from Section 5.4 in a loop) and the original performance fuzzer by Scott et al. [149] when possible since it is limited to floating-point and string logics. While there are many other fuzzers for SMT Solvers [41, 190, 187, 121], they are mostly aimed at finding errors and not performance issues.

Other general purpose fuzzers like AFL [206] and PerfFuzz [100] are built around bit-string manipulation. We attempted to use these tools but, as we suspected, neither were able to produce a well-formed input given significant amounts of resources. Unfortunately,

it is known that general purpose bit-string fuzzers do not to scale to programs with strict grammars like SMT Solvers, despite the fact that AFL has some capacity to add custom grammar [205].

Computational Environment: All experiments were performed on the Compute Canada computing service [11], a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory. Wallclock runtimes are rounded to the nearest second.

Generator: Fuzzers were set to generate benchmarks with 5 variables per sort, 5 assertions, and a maximum depth of 3 in logics that were neither just linear, just arrays, bit-vectors, nor just uninterpreted functions. Otherwise 10 variables, 10 assertions, and depth 5 was used. We use bit-widths of 64.

5.5.2 Results

Using the aforementioned experimental setup, we evaluated BanditFuzz against random fuzzing across 52 logics from the SMT-COMP '20. Tables 5.2, 5.3 summarizes our experimentation against random fuzzing across all 52. The first column in these tables denote the logic of the experiment, the second and third column denote the solver that was targeted and referenced respectively. The fourth and fifth column denotes the cumulative `PAR-2` margin across 25 runs of random fuzzing and BanditFuzz respectively. The sixth column reports improvement of BanditFuzz over random fuzzing based on the absolute difference between their `PAR-2` margins. We observe BanditFuzz to consistently outperform random fuzzing across all 52, with up to a 82.6% improvement in `PAR-2` margin in the `UFLIA` logic.

To visually illustrate the benchmark testing suites generated by BanditFuzz, we include a cactus plot on the highly industrial logic of QF_BV in Figure 5.3. A cactus plot is a visualization of a solver's performance on a benchmark suite the X-axis represents the number of benchmarks solved and the Y-axis represents time (in seconds) taken per benchmark. Every benchmark is the resultant of run a complete run of the tool. In SMT-COMP '20, the SMT Solver Bitwuzla had a strong performance, winning numerous gold medals including the QF_BV track over competing solvers CVC4, Z3, MathSAT, and Yices. However, the cactus plot clearly shows that Bitwuzla is least performant on the benchmarks produced by BanditFuzz by an extremely large margin.

In Table 5.4, we further compare against previous work by Scott et al. [149]. We baseline BanditFuzz against this work on the only two logics it supports, QF_S and QF_FP. We observe that BanditFuzz consistently outperforms against the baseline and achieves a maximum possible score in both logics, while the baseline fails to do so.

Limitations. One major limitation of performance fuzzing is the amount of time it takes to achieve results. This is partly due to the nature of the problem, making it paramount to use efficient techniques such as BanditFuzz.

As with limitations of previous work in earlier chapters, the problem encoding, or lack thereof (e.g., in this context, the SMT encoding would be states), causes substantial loss. The most pressing concern, however, is efficiency, and learning a significant policy quickly is crucial. When modeling performance fuzzing with a more sophisticated MDP (i.e., going beyond MAB), agents take substantially longer to converge due to the increased dimensionality of the underlying problem. This is particularly problematic in our context, as a single step size may take tens of minutes.

Another limitation of BanditFuzz is the longevity of the benchmarks. The benchmarks take a substantial amount of time to produce (over SMT-LIB, 200 CPU days). However, the instances discovered are often due to solver error. While this is intended, over the years, solver developers discover these issues and fix them (either due to communication from BanditFuzz experiments or externally). This poses some issues regarding the permanence of the results, as reproducing the runtime gaps on fresh builds may not result in a separation.

5.6 Case Study with SMT Solver Developers

In this section, we provide some case studies of BanditFuzz and how it enabled developers to find surprising performance deficiencies in state-of-the-art SMT solvers.

5.6.1 CVC4, Bitwuzla, and SymFPU

We contacted the developers of CVC4, Bitwuzla, and the SymFPU bit-blaster [36] for floating-point problems. While CVC4 and Bitwuzla have significantly different underlying bit-vector engines, they both utilize the SymFPU tool for bit-blasting floating-point operations. To this end, we proposed an experiment where we target both CVC4 and Bitwuzla (the target solvers) against Z3 (the reference).

The resulting benchmarks showcase performance issues in the SymFPU bit-blaster and possibly the CVC4 and Bitwuzla solvers themselves. We ran an analogous experiment to what was described in Section 5.5, with a 2400 second wallclock timeout over a 24 hour period. BanditFuzz produced 25 benchmarks that significantly separated Bitwuzla and CVC4 from Z3 on the logic of `QF_FP`. On these benchmarks that BanditFuzz produced,

Z3 had a `PAR-2` score of 3,018 seconds, while CVC4 and Bitwuzla had 91,408 seconds and 120,000 seconds respectively [4].

In discussions with Aina Niemetz and Mathias Preiner, members of CVC4 and Bitwuzla teams: "In general, the benchmarks produced by BanditFuzz can be super helpful for us to figure out what's missing in our solvers". For example, in their Bitwuzla tool, BanditFuzz found several benchmarks where the rewrite level (`-rwl`) was configured to be too high. Furthermore, Martin Brain, the author of SymFPU, said: "BanditFuzz is interesting because it gives us an abundant supply of something valuable but previously very rare; small benchmarks with significant performance differentials."

5.6.2 Z3 String Solver

We also released the BanditFuzz tool to the developers of the Z3str4 string solver [23, 24], so that they could independently use it to expose performance issues in their solver. The Z3str4 team used BanditFuzz to find performance deficiencies in experimental builds of their solver, namely Z3str4-ACF and Z3str4-NCF (the target solvers) against CVC4 and Z3seq [111]. They were able to produce thousands of benchmarks demonstrating performance separations. Mitja Kulczynski, one of the authors of Z3str4, observed: "BanditFuzz is extremely easy to use! When targeting Z3str4-NCF, BanditFuzz was able to find benchmarks in the form of disjunctions over substring operations. While this issue was already known to us, BanditFuzz provided us with a benchmark suite to improve our tool. Furthermore, when targeting Z3str4-ACF, BanditFuzz found a class of benchmarks of conjunctions of `str.at` where the solver was extremely slow. This was completely unknown to us!"

5.7 Related Work

The work that is most similar to this chapter is by Scott et al. [149]. In Sections 5.2, 5.5 of this chapter we highlight the novel contributions on of this work. Specifically, our tool uses a multi-agent RL method over the single-agent by Scott et al., and hence has a lower propensity to get stuck in local minima. Additionally, we support all of SMT-LIB, while their tool only supports floating point and strings. Another closely related tool is PerfFuzz which is a bit-string performance fuzzer. However, PerfFuzz is not grammar-aware and hence is unlikely to produce well-formed SMT formulas.

[4] Bitwuzla timed out on all benchmarks produced by BanditFuzz

Fuzzing and Fuzzing SMT Solvers: Software fuzzing is a large field of research, and we refer to the survey by Manes et al. as a basis for the current research [105]. There are tools and fuzzers for finding bugs in specific SMT theories [122, 31, 42, 41, 108, 108].

Machine Learning for Fuzzing: Bottinger et al. [33] introduce a deep Q learning algorithm for fuzzing model-free inputs. Godefroid et al. [69] use neural networks to learn an input grammar over complicated domains such as PDF and then use the learned grammar for model-guided fuzzing. Woo et al. [193] and Patil et al. [128] used MAB algorithms to select configurations of global hyper-parameters of fuzzing software. Rebert et al. [134] used MABs to select from a list of valid inputs seeds to fuzz on.

Machine Learning and SMT Solvers: Other works have leveraged machine learning to learn models relating to SMT solving performance. Healy et al. leveraged supervised learning for analyzing SMT solver performance in the context of software verification [82]. The MachSMT solver leverages machine learning SMT Solver algorithm selection [151] and the MelodySolver leverages reinforcement learning for online algorithm selection [132].

5.8 Conclusion

In this chapter, we present BanditFuzz, a performance fuzzer for SMT Solvers. BanditFuzz is the first multi-agent RL-based performance fuzzer to support all of SMT-LIB and leverages reinforcement learning to find relative performance deficiencies in state-of-the-art SMT Solvers. We evaluated BanditFuzz across 52 logics from SMT-COMP '20 targeting competition-winning solvers against runner-up solvers. We compare BanditFuzz against random fuzzing and a single-agent tool with up to a 82.6% improvement in the margin of `PAR-2` score on the `UFLIA` logic. We further provide several case studies demonstrating the utility of BanditFuzz to state-of-the-art SMT solver developers.

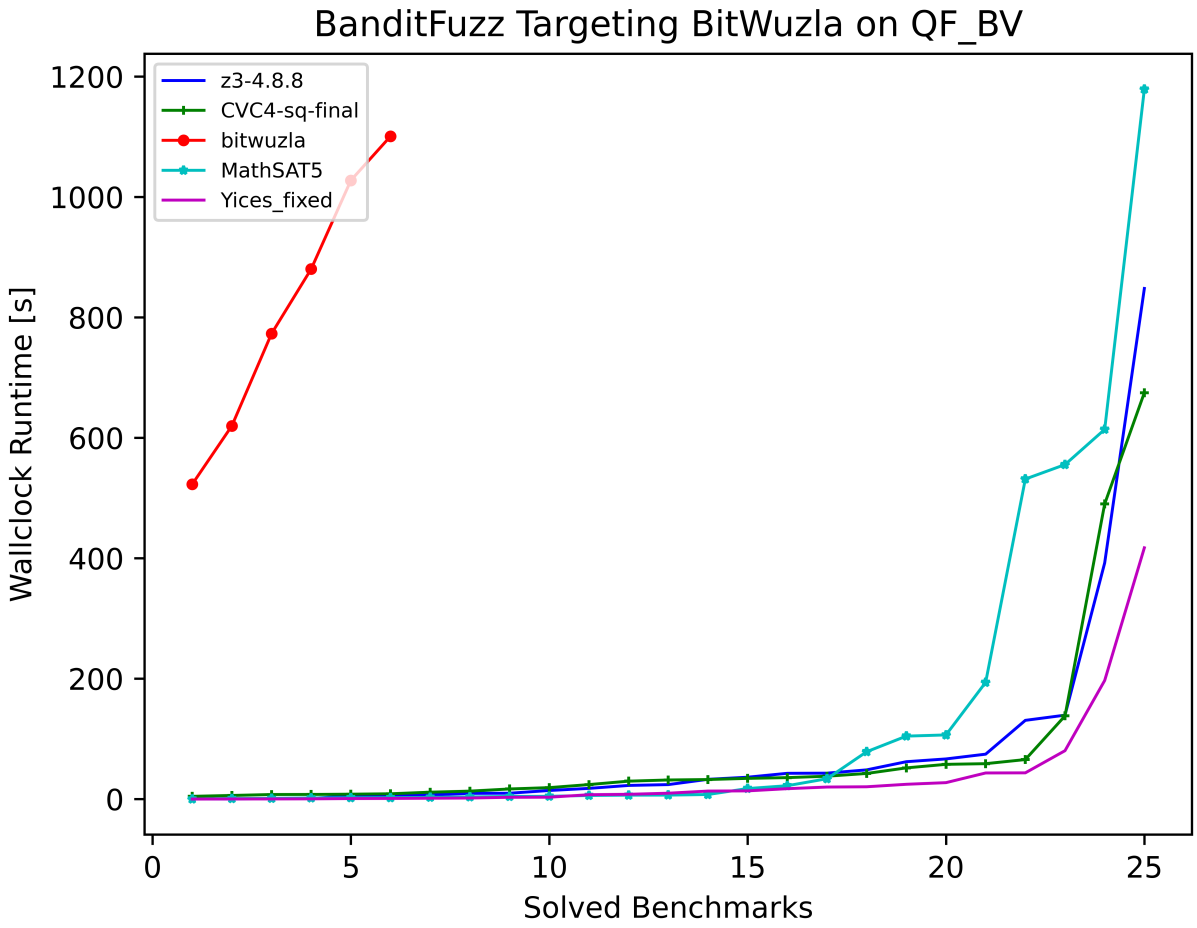


Figure 5.3: Cactus plot for targeting Bitwuzla (winner of SMT-COMP '20 in the QF_BV division) against reference runner-up solvers that competed in the division. The X-axis represents the number of benchmarks solved and the Y-axis represents time (in seconds) taken.

Chapter 6

Fuzzing Neural Network Verification Solvers

This chapter introduces `Pierce`, a versatile and extensible testing tool aimed at solvers for the neural network verification (NNV) problem. At its core, `Pierce` implements a fuzzing engine over the Open Neural Network Exchange (ONNX) – a standardized model format for deep learning and classical machine learning, and VNN-LIB – a specification standard over the input-output behavior of machine learning systems. `Pierce` supports the VNN-LIB and ONNX standard. The API of `Pierce` is designed to enable users to create a variety of software testing tools, such as performance and mutation fuzzers, as well as delta debuggers, with relative ease. For example, `Pierce` provides a rich generator for computation graphs and specifications that allows users to easily specify a wide variety of configurations, as well as mutators that ensure that mutated computation graphs are well-formed. Using `Pierce` we build a reinforcement learning (RL) driven relative performance fuzzer. Using this fuzzer, we expose performance issues in four state-of-the-art solvers, such as Marabou, ERAN, MIPVerify, and nnenum, observing up to a 13.3x times slowdown in cumulative `PAR-2` score in the target solvers relative to reference solvers. Further, we leverage `Pierce` to create a diverse benchmark suite with 10,000 competition-grade NNV instances for the community.

6.1 Motivation

Performance Fuzzing for Neural Network Verification Solvers. In recent years neural networks (NNs) have had a revolutionary impact on a variety of fields such as

computer vision [81], natural language processing [182, 204], and games [147, 161], to name just a few. Concomitant with their widespread adoption in many settings, we are also witnessing a dramatic rise in security attacks on NNs, as well as robustness and safety issues associated with NNs such as local robustness [70].

To address the above-mentioned problems, the software engineering and verification communities have developed a variety of testing [113], analysis [167], and neural network verification (NNV) tools such as Marabou [91], ERAN [163], MIPVerify [176], and nnenum [9]. While the field of NNV is still in its infancy, these NNV solvers are likely to have a huge positive impact on the robustness and security of NNs in the long run, just as SAT and SMT solvers have had on the field of software engineering and security [45, 43]. Part of the reason why SAT and SMT solvers are so impactful is that they have been subjected to a significant amount of testing, especially through the use of automated fuzzers [188, 32, 148].

Inspired by the success of debugging, testing, and analysis tools in the context of SAT/SMT solvers [14, 32, 62, 97, 115, 116, 148, 157, 185, 188, 189], we propose `Pierce`, a highly configurable testing tool for NNV solvers. Developers of NNV solvers can use `Pierce` to quickly create mutation and performance fuzzers, as well as delta debuggers. We use `Pierce` to create a performance fuzzer that can be used out-of-the-box by NNV solver developers to find performance weaknesses in their solver relative to a set of reference solvers.

Recently, it has been shown that reinforcement learning (RL) techniques can be used to develop powerful fuzzing tools that naturally take advantage of the feedback loop between the fuzzer (agent) and the testing oracle with the programs-under-test (environment) [34]. This idea is particularly relevant in the case of performance fuzzing (say in the context of solvers), where relative performance between a target and reference solver can be a powerful signal in guiding an agent to effectively modify an instance such that the relative performance difference between target and reference solvers is maximized [148, 157, 208]. To demonstrate `Pierce`'s versatility, we provide a case study implementing a relative performance fuzzer. With `Pierce`, we fuzz Marabou, ERAN, MIPVerify, and nnenum, for relative performance issues and observe up to a 13.3x times slowdown between a target and a set of reference solvers.

6.2 Pierce

In this section, we describe `Pierce`, a testing tool for ML verification (see architecture diagram in Figure 6.2).

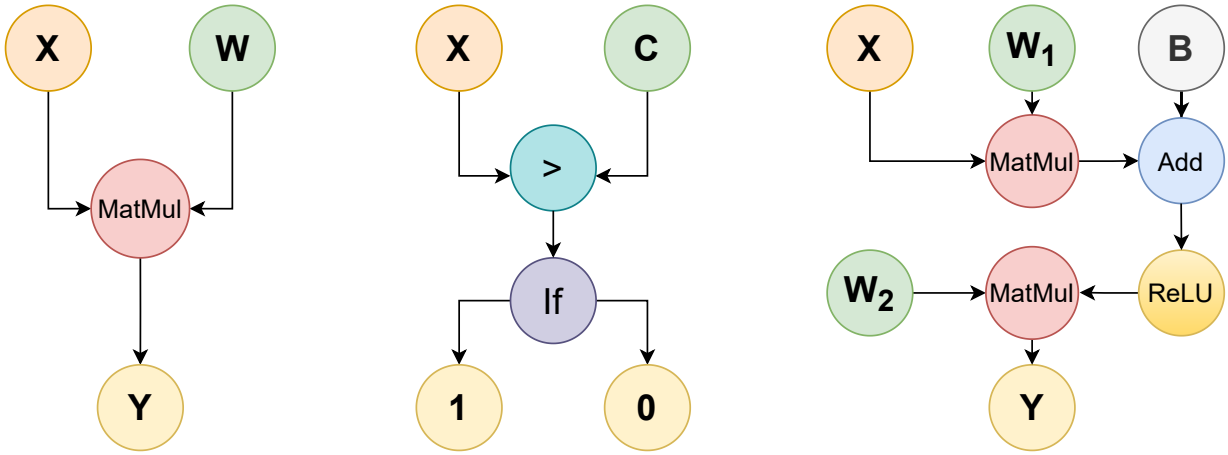


Figure 6.1: Three example computation graphs, each of which outlines a common ML algorithm. From left to right: linear regression, a depth one decision tree, a deep neural network with a single hidden layer, ReLU activation, and linear output activation. The *MatMul* stands for Matrix Multiplication.

6.2.1 Architecture Overview

See Figure 6.2 for the architecture of `Pierce`. In the context of ML verification, the input (or problem *instance*) to an ML verification system is a computation graph C and a specification ψ over the input-output behavior of C, and the output SAT (VIOLATED) or UNSAT (SAFE). The primary objective of the fuzzing engine is two-fold: generate novel instances and mutate instances, stochastically.

Generation of Computation Graphs. The generator API of `Pierce` takes as input a configuration file that includes all the parameters (approx. 550, e.g., number of parameters, depth, operator weights, etc.) for creating novel graphs, and outputs an appropriate computation graph in the ONNX format. In a computation graph, all types within the graph are tensors (i.e., a generalization of a matrix) over a primitive data type (e.g., a tensor of float32). The bulk of the logic within `Pierce`'s generator is to ensure all type, dimensionality, arity, and coarity[1] constraints are satisfied. The dimensionalities of all tensors are selected from a randomly populated set of permissible dimensions (shapes). To populate the computation graph, the generator implements a breadth-first scheme, with each visit sampling the set of operators. Cycles in G are disabled by default and not considered in this chapter. In scenarios where graph widths are significantly disproportional

[1] arity of output

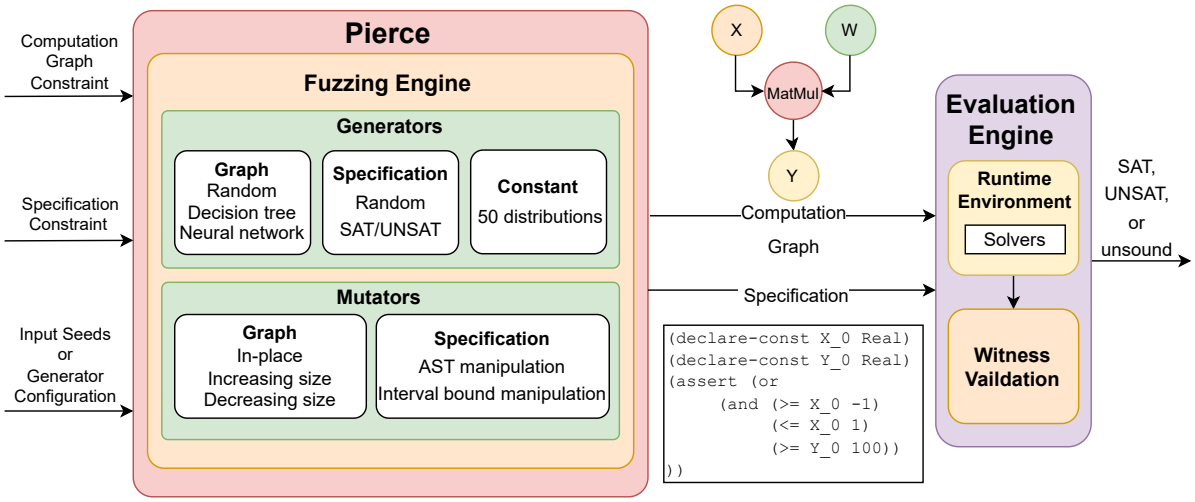


Figure 6.2: Architecture Diagram of of `Pierce` (See section 6.2). `Pierce` is comprised of a fuzzing engine that enables the generation and mutation of VNN-LIB benchmarks.

to the output, the generator imposes coarity constraints on the operation selection. Furthermore, in scenarios where dimensions cannot be matched precisely, appropriate ones are allocated. There are several size parameters in the generation process to enable the generation of graphs of all sorts of sizes (e.g., tens to billions of parameters).

The generation process supports four problem classes. First, *randomized graphs*: this is the most random and expressive problem class, representative of arbitrary machine learning programs. Second, `Pierce` can generate computation graphs resembling *decision trees*, computation graphs that contain several nested conditionals over features and constants. Third, `Pierce` can generate computation graphs resembling *random neural architectures*. This is done internally by sampling a subgraph rather than operators in the breadth-first search, where the subgraph is a composition of operators resembling a common neural layer. We leverage 20 subgraphs that denote familiar layers from `torch.nn` [126]. Finally, `Pierce` can generate *VNN-LIB graphs*, which is a subset of the previous class, such that all operators within layers are consistent with the VNN-LIB standard.

Generation of Specifications. The typical specification for a given computation graph is a set of constraints that bound the values of the input and outputs of a given computation graph. That is, a specification is either a static ϵ_x, ϵ_y pair or a component-wise $\epsilon_{ix}, \epsilon_{iy}$, (e.g., $x \in [x'_i - \epsilon_{ix}, x'_i + \epsilon_{ix}], y \in [y'_i - \epsilon_{iy}, y'_i + \epsilon_{iy}]$). The point within the domain and codomain vector space that forms the bounding is denoted by x', y', respectively. x' is always chosen randomly, while y', ϵ, ϵ' depends on the problem class. Further assertions

are added randomly. These assertions can either be linear constraints – disjunctions, conjunctions, and linear integer real arithmetic, or nonlinear. We exclusively consider linear constraints for the remainder of the chapter.

`Pierce` can generate a variety of classes of specification constraints, ranging from random specifications to specifications with known results. For example, `Pierce` can generate instances where we known that it is SAT (i.e., the specification is guaranteed to be violated). This is done by computing $y' = C(x')$ for a given computation graph C and randomly-generated input x', and generating additional constraint set A on the output of C that guarantee y' is within the bounds defined by A. Furthermore, `Pierce` can generate specifications that are UNSAT with high probability. This is done by first computing $y'' = C(x')$ for a given graph C and randomly generated input x', and then ensuring that the bounds asserted by the constraint set A on the output y'' and for a perturbation Δ with a large budget $y' = y'' + \Delta$. While the result isn't guaranteed to be UNSAT, we observe this to be highly probable for significantly large Δ and ϵ_i.

Mutating Computation Graphs and Specifications. `Pierce` has a mutator (i.e., a set of mutation operators) that ensures that a computation graph G is modified in a well-formed manner. A single mutation implements a minimal structural change. Similar to generation, the bulk of the logic is devoted to ensuring that all types, dimensionalities, arity, and coarity constraints are satisfied. For computation graph mutations, it can be done either in-place (i.e., the size of the graph remains the same) or by increasing/decreasing the size of the graph.

In-place mutations are conceptually similar to AST manipulation schemes in BanditFuzz [157]. If the mutation is applied to a computation graph C, then a node is selected, and a node of the appropriate return sort is used in an in-place swap. With incompatible coarities, subgraphs are appropriately dropped or regenerated. Increasing mutations allow for increases in dimensionality (this changes both C, ψ), insertion of a new node in C, or insertion of a new node in an asserting AST in ψ. Decreasing mutations are analogous.

A desirable property of a mutational fuzzer is satisfiability preservation, i.e., if a computation graph C is SAT (resp. UNSAT), then it remains SAT (resp. UNSAT) after mutation. Such a feature makes the fuzzer very valuable to solver developers. To see this consider the following scenario: suppose a computation graph (along with its specification) C which is known to be SAT is mutated in a satisfiability-preserving manner to C', and a solver S outputs UNSAT on C' then we know that solver S has a bug in it. This idea has been explored extensively in SMT [189]. `Pierce` includes two simple mutation procedures to support this feature. Namely, on SAT benchmarks, `Pierce` has a mutation procedure that increases the bound on the input/output of the instance and, analogously for UNSAT,

decreases it.

Constant Generation. `Pierce` supports a diverse set of seven probability distributions to generate constants. For example, `Pierce` leverages: Normal, Uniform, geometric, Beta, Gamma, Zipf, and Rayleigh. `Pierce` uses several values for the parameters of the probability distribution to compute the final set. When generating a constant tensor, the set of distributions is sampled to populate it.

Evaluation Engine. We include an evaluation engine for additional utility. The evaluation engine collects system information (e.g., wall-clock times, memory, etc.) and comes with a base solver class and example-derived classes over select VNN-LIB solvers for a quick extension. `Pierce` comes with witness validation to determine if the solver soundly determined satisfiability.

Implementation Details.`Pierce` is implemented in Python version 3.8, and uses both `onnx` and `onnxruntime` to implement all components in the fuzzing engine. `Pierce` uses `numpy` [80] for its random constant generation and all other random processes. The execution engine leverages `benchexec` [28].

6.2.2 Command Line Interfaces

`Pierce` can be invoked in three different modes depending on the CLI option - fuzz (generate a novel instance), mut (mutate a pre-existing instance), or eval (evaluate an instance on a solver).

Fuzzing. In the fuzzing mode, `Pierce` expects an argument denoting the graph problem class {rand, tree, dnn, vnnlib} for generating random computation graphs, decision tree-like graphs, neural network-like graphs, and VNN-LIB compliant graphs, respectively. Additionally, the specification problem class {rand, sat, punsat} also needs to be specified.

Mutating. In the mutating mode, `Pierce` expects an argument denoting mutation mode {graph, spec, graph-up, graph-down, preserve-sat, preserve-unsat} and a specified graph C with specification ψ for an input seed.

Evaluating. In the evaluation mode, a user specifies an instance and a solver to be evaluated. `Pierce` outputs a summary of computational resources used and the result of the solver.

Configuration. Both fuzzing and mutation processes are highly stochastic. However, significant effort has been made to ensure the user had a significant ability to adjust this

process. We provide a detailed YAML file with over 500 adjustable hyperparameters [2]. These parameters control weighted probabilities of operators, random number generators, and the generation and mutation process of the fuzzing engine.

6.2.3 Potential Use Cases

`Pierce` is engineered and designed so it can be leveraged by ML verification developers to help improve the efficiency and efficacy of their tools. We next outline some example usage.

Benchmark Generation and Testing. The generator of `Pierce`'s fuzzing system allows for benchmarks of extreme varieties and problem classes. A developer can leverage `Pierce` to create instances ranging from unit tests, regression tests, to competition difficulty instances. We release a repository of 10,000 benchmarks to the community produced by `Pierce`. The first 4,950 instances are small unit tests. These tests are broken down to test the functionality of all 187 operations of the ONNX standard. The next 4,950 instances are medium-sized instances resembling regression tests. These benchmarks are evenly distributed into their respective problem classes. The final 100 are competition-grade benchmarks that are compliant with VNN-LIB. We intend to submit these to future competitions.

Fuzzing. `Pierce` has an extensive and highly configurable fuzzing engine that can be leveraged to build several fuzzers for various objectives.

Delta Debugging. A delta-debugger is a tool to decrease the size of an error-revealing instance I such that I is minimal in size and the erroneous behavior of the program-under-test is preserved. `Pierce` has an extensive and highly configurable mutational engine that can be leveraged to build a delta debugger.

6.3 Performance Fuzzing Neural Network Verification Solvers

In this section, we present an empirical case study of `Pierce`. Specifically, we use `Pierce`'s fuzzing utility and leverage it to find relative performance slowdowns across NNV solvers by leveraging an adaption of the BanditFuzz algorithm [148].

[2]Note that these hyperparameters have reasonable default values that can make it operate in a click-of-a-button.

Targeting MIPVerify		**Targeting nnenum**	
Solver	PAR-2	Solver	PAR-2
VBS	453.5	VBS	507.2
Marabou	597.0	MIPVerify	659.0
nnenum	738.7	ERAN	786.1
ERAN	840.2	Marabou	802.0
MIPVerify	10079.8	nnenum	10057.6

Targeting ERAN		**Targeting Marabou**	
Solver	PAR-2	Solver	PAR-2
VBS	453.5	VBS	507.2
MIPVerify	605.8	nnenum	720.4
Marabou	688.3	MIPVerify	755.2
nnenum	725.5	ERAN	815.7
ERAN	10351.9	Marabou	10899.4

Table 6.1: Table of sums of PAR-2 across all experiments from the empirical evaluation (Section 6.3). The PAR-2 score of a solver on a benchmark is the wall-clock runtime if successful, otherwise twice the wall-clock runtime (lower is better). VBS denotes the virtual best solver. We observe that Pierce is able to discover instances with relative slowdowns across all considered solvers.

6.3.1 Experimental Setup

Relative Performance Fuzzing. Let T be a set of solvers to be targeted, and let R be a set of reference solvers. Let ϕ be an empirical performance margin function. The relative performance fuzzing problem seeks to solve

$$\max_{I \in L} \phi(T, R, I)$$

for a set of target solvers T, reference solvers R, and input I in the language L over solvers in T, R. In this chapter, we exclusively consider the following performance margin function

$$\phi(T, R, I) = \min_{t \in T}(\texttt{PAR-2}(t, I)) - \max_{r \in R}(\texttt{PAR-2}(r, I))$$

Bandit Formulation. For full context on the use of reinforcement learning and multi-armed bandits (MAB) in the context of performance fuzzing we refer the reader to Scott

et al. [157]. In this context, for a set of agents $\mathcal{A}$, each agent $A \in \mathcal{A}$, must have a defined action set to sample from and a global reward signal $\mathcal{R}$ across $\mathcal{A}$. For $\mathcal{R}$, we use a binary signal of whether or not the resultant benchmark produced an increase in the above-defined performance margin function. We include 5 agents in $\mathcal{A}$, controlling generator parameters of depth, width, dimensionality, and input/output sizes. We use Thompson Sampling as our base MAB algorithm [142].

Fuzzing Formulation. We consider four state-of-the-art solvers for VNN-LIB, namely, ERAN [163], Marabou [91], MIPVerify [176], and nnenum [9]. We run four fuzzing experiments, targeting each solver while using the remainder as reference solvers, and conducting each experiment 25 times to produce 25 benchmarks exposing slowdowns in each solver. Each experiment was run for 48 hours. We configure `Pierce` to generate "small" with final benchmarks ranging from 10,000 to 50,000 parameters (i.e., 100-400 neurons).

Computational Environment. All experiments were performed on the Compute Canada computing service [11], a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory. Wall-clock runtimes are rounded to the nearest second, with a wall-clock timeout of 300 seconds. Solvers were configured to run sequentially. We observe `Pierce` to consistently find relative performance slowdowns.

6.3.2 Evaluation and Analysis of Results

The cactus plot of the evaluation is presented in Figure 6.3. Table 6.1 presents the `PAR-2` all experiments. The `PAR-2` score of a solver on a benchmark is the wall-clock runtime if successful. Otherwise, twice the wall-clock runtime (lower is better). We see across all four fuzzing experiments, the target solver (in pink, last line of each legend) has a significant performance drop-off compared to the reference solvers. We include a comparison with the Virtual Best Solver (VBS). The VBS is an instance-wise minimum of runtimes which makes it representative of the best solver from a given set.

Limitations. While the `Pierce` fuzzer was designed for broad and multipurpose usage, building truly generic fuzzing tools presents challenges, and `Pierce` likely has several unseen limitations. While several fuzzing applications have been tested internally and presented in this thesis, some have not been empirically justified, such as delta-debugging.

At the time of this writing, NNV tools do not scale to the likes of LLMs, and likewise, `Pierce` is not suited for testing these sorts of instances. Another nuance is that the tech stack of `Pierce` is often very brittle and actively being developed. `Pierce` sometimes acts as a tester for these tools as opposed to NNV solvers, often finding inconsistencies and errors in existing standards such as ONNX.

6.4 Related Work

To the best of our knowledge, we are not aware of previous work on fuzzing tools for ONNX and VNN-LIB. In other domains, such as SMT, there are analysis tools that motivated us to develop `Pierce`, such as StringFuzz [32], ddsmt [97, 116, 97], and BanditFuzz [148, 157]. BanditFuzz is a fuzzing algorithm that applies Reinforcement Learning (RL) to generate inputs for Floating-Point (FP) and String SMT solvers. The RL objective of BanditFuzz is to maximize the performance margin between target and reference solvers. Counterexample-Guided Fuzzing for Neural Networks Verification is designed to discover neural network verification tools' mistakes [203]. To get more counterexamples in a sample set with a limited size and improve the performance in uncovering errors, the scope of the sample space is reduced continuously based on the generated counterexamples.

6.5 Conclusion

In this chapter, we presented `Pierce`, a flexible testing system that can be used to construct a variety of fuzzers and delta debuggers to test NNV solvers, as well as for other machine learning settings such as decision trees. To showcase the versatility and utility of `Pierce`, we implemented a relative performance fuzzer using it that in turn exposed relative performance slowdown in four state-of-the-art NNV solvers, namely, Marabou, ERAN, MIPVerify, and nnenum. We observed up to a 13.3x times slowdown in target solvers relative to reference ones. Further, using `Pierce` we created 10,000 diverse benchmarks spanning unit tests, regression tests, and competition grade benchmarks.

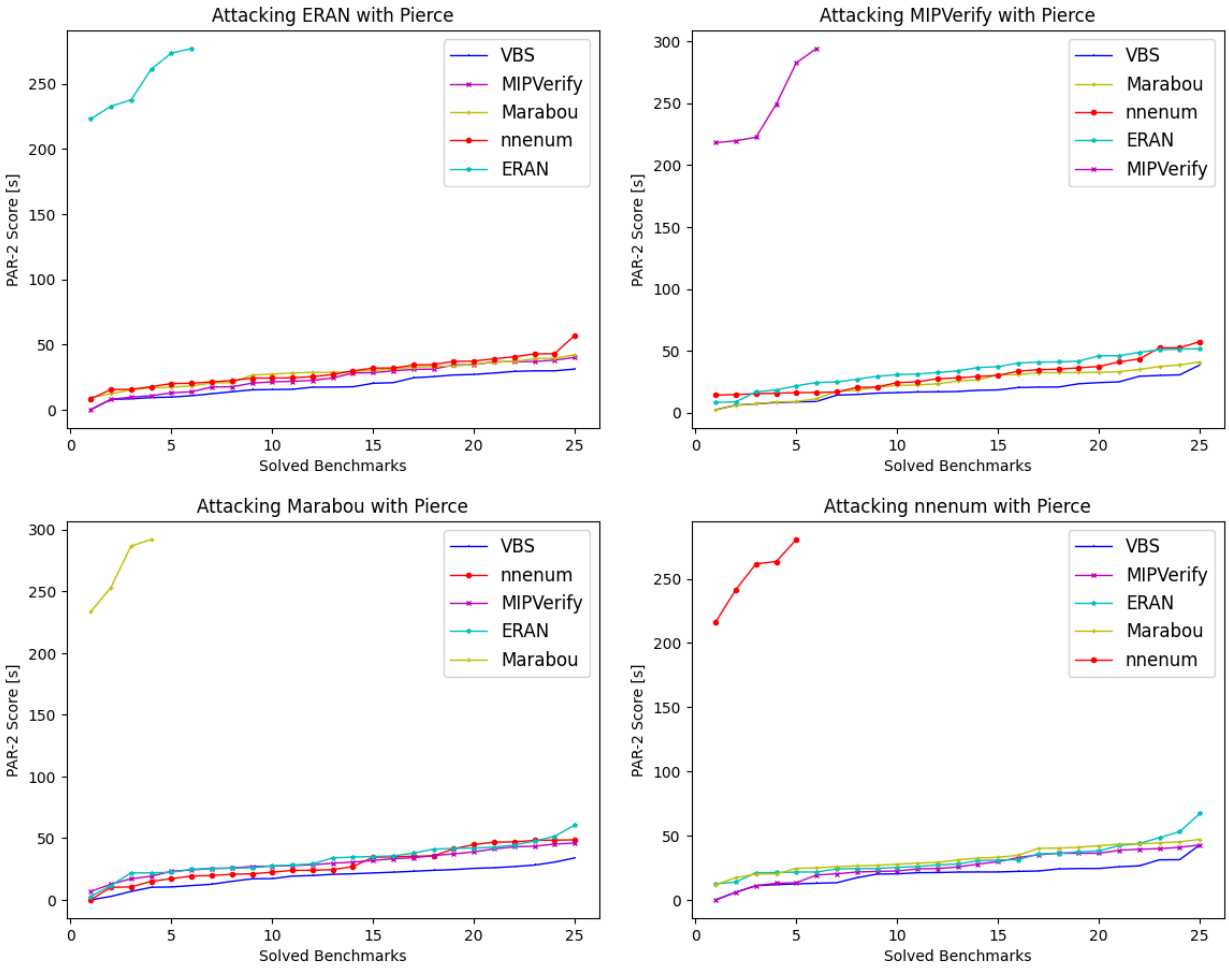


Figure 6.3: Main experimental cactus plots demonstrating `Pierce`'s ability to reveal relative performance slowdowns (Section 6.3). A cactus plot is a visualization of a solver's performance on a benchmark suite the X-axis represents the number of benchmarks solved (higher is better) and the Y-axis is the benchmark wise `PAR-2` (lower is better).

Chapter 7

Conclusions and Future Work

In this thesis, we examined the ability of machine learning techniques to advance scalability issues within logic solvers.

In Chapter 3, we introduced MachSMT, the first algorithm selection tool that spans the entirety of the SMT-LIB logics. MachSMT is designed to be user-friendly and easily modifiable by users for their specific application and SMT solvers of interest. We extensively evaluated MachSMT over several usage scenarios and empirically demonstrated its efficiency and efficacy. Using MachSMT, we observe improvement in 57 out of 119 divisions in all tracks from the SMT-COMP '19 and '20, with up to a 99.4% improvement in `PAR-2` score over the best performing solver for the QF_BVFPLRA SQ'20 division. We further evaluated MachSMT to predict a ranking and resource allocation for 23 configurations used in the cvc5 competition script and observed that MachSMT was able to solve 898 more benchmarks with up to an 93.4% improvement in `PAR-2` score. Finally, we evaluated MachSMT on network verification problems with simple domain-specific knowledge and observed an improvement of 77.3% in `PAR-2` score.

In Chapter 4, we introduced `Goose`, a meta-solver for NN verification. By meta-solver, we mean a tool containing a set of subsolvers that get adaptively called, in a sequence, based on online and offline information collected about their performance histories on a given input. `Goose` has a meta-solver architecture (Figure 4.1) and supports a wide variety of incomplete and complete solvers (some of which were implemented by us). `Goose` leverages three key meta-solving techniques, namely, adaptive algorithm selection, probabilistic satisfiability inference, and time interval deepening to implement an adaptive sequential portfolio of solvers for NN verification. We evaluate the efficacy of our `Goose` meta-solver on the VNN-COMP 2021 (resp. VNN-COMP 2022) benchmarks against 13 (resp. 11)

state-of-the-art neural network verification solvers. We observed a 37.7% (resp. 25.6%) improvement across benchmarks and solvers from VNN-COMP '21 (resp. VNN-COMP '22).

In Chapter 5, we introduced BanditFuzz, a performance fuzzer for SMT Solvers. BanditFuzz is the first multi-agent RL-based performance fuzzer to support all of SMT-LIB and leverages reinforcement learning to find relative performance deficiencies in state-of-the-art SMT Solvers. We evaluated BanditFuzz across 52 logics from SMT-COMP '20 targeting competition-winning solvers against runner-up solvers. We compare BanditFuzz against random fuzzing and a single-agent tool with up to a 82.6% improvement in the margin of `PAR-2` score on the `UFLIA` logic. We further provide several case studies demonstrating the utility of BanditFuzz to state-of-the-art SMT solver developers.

In Chapter 6, we introduced `Pierce`, a flexible testing system that can be used to construct a variety of fuzzers and delta debuggers to test NNV solvers, as well as for other machine learning settings such as decision trees. To showcase the versatility and utility of `Pierce`, we implemented a relative performance fuzzer using it that in turn exposed relative performance slowdown in four state-of-the-art NNV solvers, namely, Marabou, ERAN, MIPVerify, and nnenum. We observed up to a 13.3x times slowdown in target solvers relative to reference ones. Further, using `Pierce` we created 10,000 diverse benchmarks spanning unit tests, regression tests, and competition grade benchmarks.

Throughout this thesis, we've effectively demonstrated the use of machine learning to improve the scalability of logic solvers through the introduction of tools like MachSMT, Goose, BanditFuzz, and Pierce. As we conclude the specific advancements made, we now look ahead to potential enhancements and broader applications of these methodologies in the field of automated reasoning.

Problem Encoding. One major limitation when applying machine learning methods to automated reasoning is the problem encoding, i.e., constructing feature vectors or embeddings of an underlying problem instance. While Chapters 3, 4 provide some insights, they leave a lot to be desired. To properly represent the problem instance without loss of any information would likely require something like a graph neural network [210] which have been applied to automated reasoning with some success [174] but are not leveraged by state-of-the-art systems.

Synthetic Data. Machine learning systems are extremely data-hungry. So much so, that a recent trend within data science is data augmentation with synthetic data for more performant models [2]. All the supervised learning and reinforcement learning performed in Chapters 3, 5, 4, 6 were also likewise data-intensive and the performance of the models was internally observed to be quite sensitive the amount of quality available data. With

the use of fuzzers for being able to find interesting inputs for SMT/NNV solvers, perhaps fuzz-driven search for high quality synthetic data could be impactful on model performance.

Meta Solving. The success of `Goose` for the NN verification problem suggests that similar meta-solving techniques (i.e., adaptive sequential portfolio methods) can effective for the SAT/SMT problem as well. Hence, in the future, we plan to extend our techniques for constructing meta-solvers over SAT and SMT solvers.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[2] Mohammad Abufadda and Khalid Mansour. "A Survey of Synthetic Data Generation for Machine Learning". In: *2021 22nd International Arab Conference on Information Technology (ACIT)*. 2021, pp. 1–7. DOI: 10.1109/ACIT53391.2021.9677302.

[3] Shawkat Ali and Kate A. Smith. "On learning algorithm selection for classification". In: *Appl. Soft Comput.* 6.2 (2006), pp. 119–138. DOI: 10.1016/j.asoc.2004.12.002.

[4] Shawkat Ali and Kate A. Smith. "On learning algorithm selection for classification". In: *Appl. Soft Comput.* 6.2 (2006), pp. 119–138. DOI: 10.1016/j.asoc.2004.12.002.

[5] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. "SUNNY: a Lazy Portfolio Approach for Constraint Solving". In: *Theory Pract. Log. Program.* 14.4-5 (2014), pp. 509–524. DOI: 10.1017/S1471068414000179.

[6] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. "Bounded model checking of software using SMT solvers instead of SAT solvers". In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 69–83.

[7] Gilles Audemard and Laurent Simon. “On the glucose SAT solver”. In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001.

[8] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. “Semantic-based Automated Reasoning for AWS Access Policies using SMT”. In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by Nikolaj Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602994.

[9] Stanley Bak. “nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement”. In: *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*. Ed. by Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez. Vol. 12673. Lecture Notes in Computer Science. Springer, 2021, pp. 19–36. DOI: 10.1007/978-3-030-76384-8_2.

[10] Stanley Bak, Changliu Liu, and Taylor T. Johnson. “The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results”. In: *CoRR* abs/2109.00498 (2021). arXiv: 2109.00498. URL: https://arxiv.org/abs/2109.00498.

[11] Susan Baldwin. “Compute Canada: advancing computational research”. In: *Journal of Physics: Conference Series*. Vol. 341. 1. IOP Publishing. 2012, p. 012001.

[12] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. “Learning to Solve SMT Formulas”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 10338–10349. URL: https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html.

[13] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. “Learning to Solve SMT Formulas”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 10338–10349. URL: http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas.

[14] Tomás Balyo, Marijn Heule, and Matti Jarvisalo. "SAT competition 2016: Recent developments". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 2017.

[15] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9_24.

[16] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9_24.

[17] Haniel Barbosa, Anti Hyvärinen, and Jochen Hoenecke. *SMT-COMP 2020*. https://www.smt-comp.org/2020. 2020.

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2020.

[20] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[21] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA,*

July 14-20, 2011. Proceedings. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[22] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. "goSAT: Floating-point satisfiability as global optimization". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, pp. 11–14. DOI: 10.23919/FMCAD.2017.8102235.

[23] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. "Z3str3: A string solver with theory-aware heuristics". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, pp. 55–59. DOI: 10.23919/FMCAD.2017.8102241.

[24] Murphy Berzish, Federico Mora, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. "Z3str4 String Solver: System Description". In: *SMT-COMP 2020* (2020).

[25] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. *The SCIP Optimization Suite 8.0*. ZIB-Report 21-41. Zuse Institute Berlin, 2021. URL: http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

[26] Ksenia Bestuzheva, Mathieu Besançon, Weikun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros M. Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. "Enabling Research through the SCIP Optimization Suite 8.0". In: *ACM Trans. Math. Softw.* 49.2 (2023), 22:1–22:21. DOI: 10.1145/3585516. URL: https://doi.org/10.1145/3585516.

[27] Dirk Beyer and Matthias Dangl. “Strategy Selection for Software Verification Based on Boolean Features - A Simple but Effective Approach”. In: Lecture Notes in Computer Science 11245 (2018). Ed. by Tiziana Margaria and Bernhard Steffen, pp. 144–159. DOI: 10.1007/978-3-030-03421-4_11.

[28] Dirk Beyer, Stefan Löwe, and Philipp Wendler. “Reliable benchmarking: requirements and solutions”. In: *Int. J. Softw. Tools Technol. Transf.* 21.1 (2019), pp. 1–29. DOI: 10.1007/s10009-017-0469-y.

[29] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. “Everest: Towards a Verified, Drop-in Replacement of HTTPS”. In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 1:1–1:12. DOI: 10.4230/LIPIcs.SNAPL.2017.1. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2017.1.

[30] Armin Biere. “Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018”. In: *Proceedings of SAT Competition* 14 (2017), pp. 316–336.

[31] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. “StringFuzz: A Fuzzer for String Solvers”. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 45–51. DOI: 10.1007/978-3-319-96142-2_6.

[32] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. “StringFuzz: A Fuzzer for String Solvers”. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 45–51. DOI: 10.1007/978-3-319-96142-2_6.

[33] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. “Deep Reinforcement Fuzzing”. In: (2018), pp. 116–122. DOI: 10.1109/SPW.2018.00026.

[34] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. "Deep Reinforcement Fuzzing". In: *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 2018, pp. 116–122. DOI: 10.1109/SPW.2018.00026.

[35] Marin Bougeret, Pierre-François Dutot, Alfredo Goldman, Yanik Ngoko, and Denis Trystram. "Combining multiple heuristics on discrete resources". In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160879.

[36] Martin Brain. *SymFPU*. 2019. URL: https://github.com/martin-cs/symfpu/ (visited on 01/01/0201).

[37] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. "Deciding floating-point logic with abstract conflict driven clause learning". In: *Formal Methods Syst. Des.* 45.2 (2014), pp. 213–245. DOI: 10.1007/s10703-013-0203-7.

[38] Martin Brain, Florian Schanda, and Youcheng Sun. "Building Better Bit-Blasting for Floating-Point Problems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 79–98. DOI: 10.1007/978-3-030-17462-0_5.

[39] Martin Brain, Florian Schanda, and Youcheng Sun. "Building Better Bit-Blasting for Floating-Point Problems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 79–98. DOI: 10.1007/978-3-030-17462-0_5.

[40] Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T. Johnson, and Changliu Liu. "First three years of the international verification of neural networks competition (VNN-COMP)". In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 329–339. DOI: 10.1007/S10009-023-00703-4. URL: https://doi.org/10.1007/s10009-023-00703-4.

[41] Robert Brummayer and Armin Biere. "Fuzzing and delta-debugging SMT solvers". In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. ACM. 2009, pp. 1–5.

[42] Alexandra Bugariu and Peter Müller. "Automatically Testing String Solvers". In: *International Conference on Software Engineering (ICSE), 2020*. ETH Zurich. 2020.

[43] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[44] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008), 10:1–10:38. DOI: 10.1145/1455518.1455522.

[45] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008), 10:1–10:38. DOI: 10.1145/1455518.1455522.

[46] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. "HornDroid: Practical and sound static analysis of Android applications by SMT solving". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 47–62.

[47] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[48] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[49] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[50] Yudong Chen and Martin J. Wainwright. "Fast low-rank estimation by projected gradient descent: General statistical and algorithmic guarantees". In: *CoRR* abs/1509.03025 (2015). arXiv: 1509.03025. URL: http://arxiv.org/abs/1509.03025.

[51] François Chollet et al. *Keras*. https://keras.io. 2015.

[52] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An Interpolating SMT Solver". In: *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. Ed. by Alastair F. Donaldson and David Parker. Vol. 7385. Lecture Notes in Computer Science. Springer, 2012, pp. 248–254. DOI: 10.1007/978-3-642-31759-0_19. URL: https://doi.org/10.1007/978-3-642-31759-0_19.

[53] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 93–107. DOI: 10.1007/978-3-642-36742-7_7.

[54] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[55] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. "SMT-based bounded model checking for embedded ANSI-C software". In: *IEEE Transactions on Software Engineering* 38.4 (2011), pp. 957–974.

[56] IBM ILOG Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[57] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[58] Fabien Duchene. "Fuzz in the Dark: Genetic Algorithm for Black-Box Fuzzing". In: *Black-Hat*. 2013.

[59] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[60] ABKFM Fleury and Maximilian Heisinger. "Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020". In: *SAT COMPETITION* 2020 (2020), p. 50.

[61] Yoav Freund, Robert Schapire, and Naoki Abe. "A short introduction to boosting". In: *Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999), p. 1612.

[62] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. "SAT Competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103572.

[63] Zhoulai Fu and Zhendong Su. "XSat: A Fast Floating-Point Satisfiability Solver". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 187–209. DOI: 10.1007/978-3-319-41540-6_11.

[64] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. "ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 209–213. DOI: 10.1007/978-3-030-17502-3_15.

[65] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 519–531. DOI: 10.1007/978-3-540-73368-3_52.

[66] Ian P. Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. "Learning When to Use Lazy Learning in Constraint Solving". In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*. Ed. by Helder Coelho,

Rudi Studer, and Michael J. Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 873–878. DOI: 10.3233/978-1-60750-606-5-873.

[67] Ralf Gerlich and Christian R Prause. "Optimizing the Parameters of an Evolutionary Algorithm for Fuzzing and Test Data Generation". In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2020, pp. 338–345.

[68] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "SAGE: whitebox fuzzing for security testing". In: *Commun. ACM* 55.3 (2012), pp. 40–44. DOI: 10.1145/2093548.2093564.

[69] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: machine learning for input fuzzing". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 50–59. DOI: 10.1109/ASE.2017.8115618.

[70] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6572.

[71] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. "The Boogie Verification Debugger (Tool Paper)". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 407–414. DOI: 10.1007/978-3-642-24690-6_28.

[72] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy A. Mann, and Pushmeet Kohli. "On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models". In: *CoRR* abs/1810.12715 (2018). arXiv: 1810.12715. URL: http://arxiv.org/abs/1810.12715.

[73] Dario Guidotti, Stefano Demarchi, Armando Tacchella, and Luca Pulina. *The Verification of Neural Networks Library (VNN-LIB)*. https://www.vnnlib.org. 2023.

[74] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. "Program analysis as constraint solving". In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 281–292.

[75] Arjun K Gupta and Saralees Nadarajah. *Handbook of beta distribution and its applications*. CRC press, 2004.

[76] Harshit Gupta, Kyong Hwan Jin, Ha Q. Nguyen, Michael T. McCann, and Michael Unser. "CNN-Based Projected Gradient Descent for Consistent CT Image Reconstruction". In: *IEEE Trans. Medical Imaging* 37.6 (2018), pp. 1440–1453. DOI: 10.1109/TMI.2018.2832656.

[77] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com.

[78] Liana Hadarean, Anti Hyvärinen, Aina Niemetz, and Giles Reger. *SMT-COMP 2019*. https://www.smt-comp.org/2019. 2019.

[79] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[80] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. URL: https://doi.org/10.1109/CVPR.2016.90.

[82] Andrew Healy, Rosemary Monahan, and James F. Power. "Predicting SMT Solver Performance for Software Verification". In: *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016*. Ed. by Catherine Dubois, Paolo Masci, and Dominique Méry. Vol. 240. EPTCS. 2016, pp. 20–37. DOI: 10.4204/EPTCS.240.2.

[83] Patrick Henriksen and Alessio R. Lomuscio. "Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search". In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by Giuseppe

De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2513–2520. DOI: 10.3233/FAIA200385.

[84] Jan Hula, David Mojzísek, and Mikolás Janota. "Graph Neural Networks for Scheduling of SMT Solvers". In: *33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021, Washington, DC, USA, November 1-3, 2021*. IEEE, 2021, pp. 447–451. DOI: 10.1109/ICTAI52525.2021.00072. URL: https://doi.org/10.1109/ICTAI52525.2021.00072.

[85] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. "Proteus: A Hierarchical Portfolio of Solvers and Transformations". In: *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*. Ed. by Helmut Simonis. Vol. 8451. Lecture Notes in Computer Science. Springer, 2014, pp. 301–317. DOI: 10.1007/978-3-319-07046-9_22.

[86] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. "Automated analysis and debugging of network connectivity policies". In: *Microsoft Research* (2014), pp. 1–11.

[87] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and detecting real-world performance bugs". In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 77–88.

[88] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. "SMT-based false positive elimination in static program analysis". In: *International Conference on Formal Engineering Methods*. Springer. 2012, pp. 316–331.

[89] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 97–117. DOI: 10.1007/978-3-319-63387-9_5.

[90] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 97–117. DOI: 10.1007/978-3-319-63387-9_5. URL: https://doi.org/10.1007/978-3-319-63387-9_5.

[91] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 443–452. DOI: 10.1007/978-3-030-25540-4_26.

[92] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[93] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-based model checking for recursive programs". In: *Formal Methods in System Design* 48.3 (2016), pp. 175–205.

[94] Lars Kotthoff. "Algorithm Selection for Combinatorial Search Problems: A Survey". In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*. Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi. Vol. 10101. Lecture Notes in Computer Science. Springer, 2016, pp. 149–190. DOI: 10.1007/978-3-319-50137-6_7.

[95] Lars Kotthoff. "Algorithm Selection for Combinatorial Search Problems: A Survey". In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*. Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi. Vol. 10101. Lecture Notes in Computer Science. Springer, 2016, pp. 149–190. DOI: 10.1007/978-3-319-50137-6_7.

[96] Lars Kotthoff, Ian P. Gent, and Ian Miguel. "An Evaluation of Machine Learning in Algorithm Selection for Search Problems". In: *AI Commun.* 25.3 (2012), pp. 257–270. DOI: 10.3233/AIC-2012-0533. URL: https://doi.org/10.3233/AIC-2012-0533.

[97] Gereon Kremer, Aina Niemetz, and Mathias Preiner. "ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 231–242. DOI: 10.1007/978-3-030-81688-9_11.

[98] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Comput.* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[99] K. Rustan M. Leino. "Automating Theorem Proving with SMT". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 2–16. DOI: 10.1007/978-3-642-39634-2_2.

[100] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. "Perffuzz: Automatically generating pathological inputs". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 254–265.

[101] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 123–140. DOI: 10.1007/978-3-319-40970-2_9.

[102] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 123–140. DOI: 10.1007/978-3-319-40970-2_9. URL: https://doi.org/10.1007/978-3-319-40970-2_9.

[103] Giovanni M. Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. "DASH: Dynamic Approach for Switching Heuristics". In: *CoRR* abs/1307.4689 (2013). arXiv: 1307.4689. URL: http://arxiv.org/abs/1307.4689.

[104] Yuri Malitsky. "Evolving Instance-Specific Algorithm Configuration". In: *Instance-Specific Algorithm Configuration*. Springer, 2014, pp. 93–105. ISBN: 978-3-319-11229-9. DOI: 10.1007/978-3-319-11230-5. URL: https://doi.org/10.1007/978-3-319-11230-5.

[105] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "Fuzzing: Art, Science, and Engineering". In: *arXiv preprint arXiv:1812.00140* (2018).

[106] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. "Ankou: guiding grey-box fuzzing towards combinatorial difference". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1024–1036.

[107] Guido Manfredi and Yannick Jestin. "An introduction to ACAS Xu and the challenges ahead". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE. 2016, pp. 1–9.

[108] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing". In: *arXiv preprint arXiv:2004.05934* (2020).

[109] Martin Suda Marijn Heule Matti Järvisalo. *SAT Race 2019*. 2019. URL: http://sat-race-2019.ciirc.cvut.cz/ (visited on 06/01/2019).

[110] Andrew W Moore. "Cross-validation for detecting and preventing overfitting". In: *School of Computer Science Carneigie Mellon University* (2001).

[111] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[112] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[113] Nagisetty, Vineel. "Domain Knowledge Guided Testing and Training of Neural Networks". MA thesis. University of Waterloo, 2021. URL: http://hdl.handle.net/10012/17272.

[114] Saeed Nejati, Ludovic Le Frioux, and Vijay Ganesh. "A Machine Learning Based Splitting Heuristic for Divide-and-Conquer Solvers". In: *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*. Ed. by Helmut Simonis.

Vol. 12333. Lecture Notes in Computer Science. Springer, 2020, pp. 899–916. DOI: 10.1007/978-3-030-58475-7_52.

[115] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. "Impact of community structure on SAT solver performance". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2014, pp. 252–268.

[116] Aina Niemetz and Armin Biere. "ddSMT: A Delta Debugger for the SMT-LIB v2 Format". In: *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013*. 2013, pp. 36–45.

[117] Aina Niemetz and Mathias Preiner. "Bitwuzla at the SMT-COMP 2020". In: *CoRR* abs/2006.01621 (2020). arXiv: 2006.01621. URL: https://arxiv.org/abs/2006.01621.

[118] Aina Niemetz and Mathias Preiner. "Bitwuzla at the SMT-COMP 2020". In: *CoRR* abs/2006.01621 (2020). arXiv: 2006.01621. URL: https://arxiv.org/abs/2006.01621.

[119] Aina Niemetz and Mathias Preiner. "Ternary Propagation-Based Local Search for more Bit-Precise Reasoning". In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 214–224. DOI: 10.34727/2020/ISBN.978-3-85448-042-6_29. URL: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29.

[120] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: 10.3233/sat190101.

[121] Aina Niemetz, Mathias Preiner, and Armin Biere. "Model-based API testing for SMT solvers". In: *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT*. 2017, pp. 24–28.

[122] Aina Niemetz, Mathias Preiner, and Armin Biere. "Model-Based API Testing for SMT Solvers". In: *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017*. Ed. by Martin Brain and Liana Hadarean. 2017, 10 pages.

[123] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. "Syntax-Guided Rewrite Rule Enumeration for SMT Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 279–297. DOI: 10.1007/978-3-030-24258-9_20. URL: https://doi.org/10.1007/978-3-030-24258-9_20.

[124] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. "Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving". In: *Irish conference on artificial intelligence and cognitive science*. 2008, pp. 210–216. URL: https://homepages.laas.fr/ehebrard/papers/aics2008.pdf.

[125] Corina S. Pasareanu and Willem Visser. "A survey of new trends in symbolic execution for software testing and analysis". In: *Int. J. Softw. Tools Technol. Transf.* 11.4 (2009), pp. 339–353. DOI: 10.1007/s10009-009-0118-1.

[126] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 8024–8035. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[127] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[128] Ketan Patil and Aditya Kanade. "Greybox fuzzing as a contextual bandits problem". In: *arXiv preprint arXiv:1806.03806* (2018).

[129] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[130] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=2078195.

[131] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. "MedleySolver: Online SMT Algorithm Selection". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 453–470. DOI: 10.1007/978-3-030-80223-3_31. URL: https://doi.org/10.1007/978-3-030-80223-3_31.

[132] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. "MedleySolver: Online SMT Algorithm Selection". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 453–470. DOI: 10.1007/978-3-030-80223-3_31.

[133] Luca Pulina and Armando Tacchella. "A Multi-engine Solver for Quantified Boolean Formulas". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 574–589. DOI: 10.1007/978-3-540-74970-7_41.

[134] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. "Optimizing Seed Selection for Fuzzing". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 861–875. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert.

[135] Alexander Reinefeld and T. Anthony Marsland. "Enhanced Iterative-Deepening Search". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 16.7 (1994), pp. 701–710. DOI: 10.1109/34.297950.

[136] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu.

Cham: Springer International Publishing, 2015, pp. 198–216. ISBN: 978-3-319-21668-3.

[137] John R. Rice. "The Algorithm Selection Problem". In: *Adv. Comput.* 15 (1976), pp. 65–118. DOI: 10.1016/S0065-2458(08)60520-3.

[138] John R. Rice. "The Algorithm Selection Problem". In: *Adv. Comput.* 15 (1976), pp. 65–118. DOI: 10.1016/S0065-2458(08)60520-3.

[139] Cedric Richter and Heike Wehrheim. "PeSCo: Predicting Sequential Combinations of Verifiers - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 229–233. DOI: 10.1007/978-3-030-17502-3_19.

[140] Jussi Rintanen. "Madagascar: Scalable planning with sat". In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014).

[141] Juan Diego Rodríguez, Aritz Pérez Martínez, and José Antonio Lozano. "Sensitivity Analysis of k-Fold Cross Validation in Prediction Error Estimation". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.3 (2010), pp. 569–575. DOI: 10.1109/TPAMI.2009.187.

[142] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. "A Tutorial on Thompson Sampling". In: *Found. Trends Mach. Learn.* 11.1 (2018), pp. 1–96. DOI: 10.1561/2200000070.

[143] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. "A tutorial on thompson sampling". In: *Foundations and Trends® in Machine Learning* 11.1 (2018), pp. 1–96.

[144] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. "A review of machine learning applications in fuzzing". In: *arXiv preprint arXiv:1906.11133* (2019).

[145] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. "A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 9835–9846.

[146] Rocco Salvia, Laura Titolo, Marco A. Feliú, Mariano M. Moscato, César A. Muñoz, and Zvonimir Rakamaric. "A Mixed Real and Floating-Point Solver". In: *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. Lecture Notes in Computer Science. Springer, 2019, pp. 363–370. DOI: 10.1007/978-3-030-20652-9_25.

[147] Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. "Solving Checkers". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, 2005, pp. 292–297. URL: http://ijcai.org/Proceedings/05/Papers/0515.pdf.

[148] Joseph Scott, Federico Mora, and Vijay Ganesh. "BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers". In: *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Ed. by Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel. Vol. 12549. Lecture Notes in Computer Science. Springer, 2020, pp. 68–86. DOI: 10.1007/978-3-030-63618-0_5. URL: https://doi.org/10.1007/978-3-030-63618-0_5.

[149] Joseph Scott, Federico Mora, and Vijay Ganesh. "BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning". In: *UWSpace. http://hdl.handle.net/10012/15753* (2020).

[150] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. "Algorithm selection for SMT". In: *Int. J. Softw. Tools Technol. Transf.* 25.2 (2023), pp. 219–239. DOI: 10.1007/s10009-023-00696-0. URL: https://doi.org/10.1007/s10009-023-00696-0.

[151] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. "MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 303–325. DOI: 10.1007/978-3-030-72013-1_16.

[152] Joseph Scott, Guanting Pan, Elias B. Khali, and Vijay Ganesh. "Pierce: A Testing Infrastructure for Neural Network Verification Tools". In: Submitted to VSTTE 2023. 2023.

[153] Joseph Scott, Guanting Pan, Elias B. Khalil, and Vijay Ganesh. "Goose: A Meta-Solver for Deep Neural Network Verification". In: *Proceedings of the 20th International Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*. Ed. by David Déharbe and Antti E. J. Hyvärinen. Vol. 3185. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 99–113. URL: https://ceur-ws.org/Vol-3185/extended678.pdf.

[154] Joseph Scott, Guanting Pan, Elias B. Khalil, and Vijay Ganesh. "Pierce: A Testing Infrastructure for Machine Learning Verification Tools". In: *URL https://guantingpan.github.io/pie* (2022).

[155] Joseph Scott, Maysum Panju, and Vijay Ganesh. "LGML: Logic Guided Machine Learning (Student Abstract)". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 13909–13910. URL: https://aaai.org/ojs/index.php/AAAI/article/view/7227.

[156] Joseph Scott, Pascal Poupart, and Vijay Ganesh. "An Algorithm Selection Approach for QF_FP Solvers". In: *17th International Workshop on Satisfiability Modulo Theories*. 2019.

[157] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. "BanditFuzz: Fuzzing SMT Solvers with Multi-agent Reinforcement Learning". In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 103–121. DOI: 10.1007/978-3-030-90870-6_6. URL: https://doi.org/10.1007/978-3-030-90870-6_6.

[158] Daniel Selsam and Nikolaj S. Bjørner. "NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions". In: *CoRR* abs/1903.04671 (2019). arXiv: 1903.04671. URL: http://arxiv.org/abs/1903.04671.

[159] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. "Learning a SAT Solver from Single-Bit Supervision". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=HJMC_iA5tm.

[160] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. "DNNV: A Framework for Deep Neural Network Verification". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 137–150. DOI: 10.1007/978-3-030-81685-8_6.

[161] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: 10.1038/NATURE16961. URL: https://doi.org/10.1038/nature16961.

[162] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. "An abstract domain for certifying neural networks". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 41:1–41:30. DOI: 10.1145/3290354.

[163] Gagandeep Singh, Jonathan Maurer, Christoph Müller, Matthew Mirman, Timon Gehr, Adrian Hoffmann, Petar Tsankov, Dana Drachsler Cohen, Markus Püschel, and Martin Vechev. "ETH robustness analyzer for neural networks (ERAN), 2020". In: *URL https://github.com/eth-sri/eran* (2022).

[164] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramèr, Atul Prakash, and Tadayoshi Kohno. "Physical Adversarial Examples for Object Detectors". In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. Ed. by Christian Rossow and Yves Younan. USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/eykholt.

[165] Eunhye Song, Barry L. Nelson, and Jeremy Staum. "Shapley Effects for Global Sensitivity Analysis: Theory and Computation". In: *SIAM/ASA Journal on Uncertainty Quantification* 4.1 (2016), pp. 1060–1083. DOI: 10.1137/15M1048070. eprint: https://doi.org/10.1137/15M1048070. URL: https://doi.org/10.1137/15M1048070.

[166] Niklas Sorensson and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization". In: *SAT* 2005.53 (2005), pp. 1–2.

[167] Matthew Sotoudeh, Zhe Tao, and Aditya V. Thakur. "SyReNN: A tool for analyzing deep neural networks". In: *Int. J. Softw. Tools Technol. Transf.* 25.2 (2023), pp. 145–165. DOI: 10.1007/S10009-023-00695-1. URL: https://doi.org/10.1007/s10009-023-00695-1.

[168] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. "From program verification to program synthesis". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2010, pp. 313–326.

[169] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. "StarExec: A Cross-Community Infrastructure for Logic Solving". In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562. Lecture Notes in Computer Science. Springer, 2014, pp. 367–373. DOI: 10.1007/978-3-319-08587-6_28.

[170] Geoff Sutcliffe. "The TPTP Problem Library and Associated Infrastructure - From CNF to TH0, TPTP v6.4.0". In: *J. Autom. Reason.* 59.4 (2017), pp. 483–502. DOI: 10.1007/s10817-017-9407-7.

[171] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[172] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[173] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[174] Anderson R. Tavares, Pedro H. C. Avelar, João M. Flach, Márcio Nicolau, Luís C. Lamb, and Moshe Y. Vardi. "Understanding Boolean Function Learnability on Deep Neural Networks". In: *CoRR* abs/2009.05908 (2020). arXiv: 2009.05908. URL: https://arxiv.org/abs/2009.05908.

[175] Kevin Tierney and Yuri Malitsky. "An Algorithm Selection Benchmark of the Container Pre-marshalling Problem". In: *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*. Ed. by Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion. Vol. 8994. Lecture Notes in Computer Science. Springer, 2015, pp. 17–22. DOI: 10.1007/978-3-319-19084-6_2.

[176] Vincent Tjeng and Russ Tedrake. "Verifying Neural Networks with Mixed Integer Programming". In: *CoRR* abs/1711.07356 (2017). arXiv: 1711.07356. URL: http://arxiv.org/abs/1711.07356.

[177] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=HyGIdiRqtm.

[178] B.A. Trakhtenbrot. "A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms". In: *Annals of the History of Computing* 6.4 (1984), pp. 384–400. DOI: 10.1109/MAHC.1984.10036.

[179] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. "MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance". In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 441–456. DOI: 10.1007/978-3-540-71070-7_37.

[180] Mauro Vallati, Lukás Chrpa, and Diane E. Kitchin. "Portfolio-based planning: State of the art, common practice and open challenges". In: *AI Commun.* 28.4 (2015), pp. 717–733. DOI: 10.3233/AIC-150671.

[181] Mauro Vallati, Lukás Chrpa, and Diane E. Kitchin. "Portfolio-based planning: State of the art, common practice and open challenges". In: *AI Commun.* 28.4 (2015), pp. 717–733. DOI: 10.3233/AIC-150671.

[182] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[183] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification". In: *Advances in Neural Information Processing Systems* 34 (2021).

[184] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. "The SMT Competition 2015-2018". In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 221–259. DOI: 10.3233/SAT190123. URL: https://doi.org/10.3233/SAT190123.

[185] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. "The SMT Competition 2015-2018". In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 221–259. URL: https://doi.org/10.3233/SAT190123.

[186] Sheng-Han Wen, Wei-Loon Mow, Wei-Ning Chen, Chien-Yuan Wang, and Hsu-Chun Hsiao. *Enhancing Symbolic Execution by Machine Learning Based Solver Selection*. Jan. 2019. DOI: 10.14722/bar.2019.23080.

[187] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–25.

[188] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 193:1–193:25. DOI: 10.1145/3428261.

[189] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 718–730. DOI: 10.1145/3385412.3385985.

[190] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion." In: *PLDI*. 2020, pp. 718–730.

[191] *Wolfram Alpha: Computational Intelligence*. Accessed: 2023-05-25. 2023. URL: https://www.wolframalpha.com/.

[192] D.H. Wolpert and W.G. Macready. "No free lunch theorems for optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893.

[193] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. "Scheduling black-box mutational fuzzing". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM, 2013, pp. 511–522. DOI: 10.1145/2508859.2516736.

[194] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. "Automatic perturbation analysis for scalable certified robustness and beyond". In: *Advances in Neural Information Processing Systems* 33 (2020).

[195] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. "Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=nVZtXBI6LNn.

[196] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. ": The Design and Analysis of an Algorithm Portfolio for SAT". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 712–727. DOI: 10.1007/978-3-540-74970-7_50.

[197] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 228–241. DOI: 10.1007/978-3-642-31612-8_18.

[198] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 712–727. DOI: 10.1007/978-3-540-74970-7_50.

[199] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *J. Artif. Intell. Res.* 32 (2008), pp. 565–606. DOI: 10.1613/jair.2490.

[200] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *J. Artif. Intell. Res.* 32 (2008), pp. 565–606. DOI: 10.1613/jair.2490.

[201] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla2009: an automatic algorithm portfolio for SAT". In: *SAT* 4 (2009), pp. 53–55.

[202] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla2012: Improved algorithm selection based on cost-sensitive classification models". In: *Proceedings of SAT Challenge* (2012), pp. 57–58.

[203] Gaolei Yi, Xinyi Wang, and Yichen Wang. "An Empirical Study of Counterexample-Guided Fuzzing for Neural Networks Verification". In: *7th International Conference on Dependable Systems and Their Applications, DSA 2020, Xi'an, China, November 28-29, 2020*. IEEE, 2020, pp. 108–113. DOI: 10.1109/DSA51864.2020.00022.

[204] Kazuki Yoshizoe and Martin Müller. "Computer Go". In: *Encyclopedia of Computer Graphics and Games*. Ed. by Newton Lee. Springer, 2019. DOI: 10.1007/978-3-319-08234-9_20-1. URL: https://doi.org/10.1007/978-3-319-08234-9_20-1.

[205] Michal Zalewski. *afl-fuzz: making up grammar with a dictionary in hand*. 2015. URL: https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html (visited on 01/01/2015).

[206] Michal Zalewski. *American fuzzy lop*. 2015.

[207] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 4939–4948. URL: https://arxiv.org/pdf/1811.00866.pdf.

[208] Yao Zhang, Xiaofei Xie, Yi Li, Yun Lin, Sen Chen, Yang Liu, and Xiaohong Li. "Demystifying Performance Regressions in String Solvers". In: *IEEE Transactions on Software Engineering* (2022). DOI: 10.1109/TSE.2022.3168373.

[209] Yao Zhang, Xiaofei Xie, Yi Li, Yun Lin, Sen Chen, Yang Liu, and Xiaohong Li. "Demystifying Performance Regressions in String Solvers". In: *IEEE Trans. Software Eng.* 49.3 (2023), pp. 947–961. DOI: 10.1109/TSE.2022.3168373. URL: https://doi.org/10.1109/TSE.2022.3168373.

[210] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph neural networks: A review of methods and applications". In: *AI open* 1 (2020), pp. 57–81.