Shuvendu K. Lahiri
Chao Wang (Eds.)

LNCS 12224

Computer Aided Verification

32nd International Conference, CAV 2020
Los Angeles, CA, USA, July 21–24, 2020
Proceedings, Part I

Part I

OPEN ACCESS

Springer

Shuvendu K. Lahiri · Chao Wang (Eds.)

Computer Aided Verification

32nd International Conference, CAV 2020
Los Angeles, CA, USA, July 21–24, 2020
Proceedings, Part I

Springer

Editors
Shuvendu K. Lahiri
Microsoft Research Lab
Redmond, WA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-53287-1 ISBN 978-3-030-53288-8 (eBook)
https://doi.org/10.1007/978-3-030-53288-8

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

It was our privilege to serve as the program chairs for CAV 2020, the 32nd International Conference on Computer-Aided Verification. CAV 2020 was held as a virtual conference during July 21–24, 2020. The tutorial day was on July 20, 2020, and the pre-conference workshops were held during July 19–20, 2020. Due to the coronavirus disease (COVID-19) outbreak, all events took place online.

CAV is an annual conference dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The primary focus of CAV is to extend the frontiers of verification techniques by expanding to new domains such as security, quantum computing, and machine learning. This puts CAV at the cutting edge of formal methods research, and this year's program is a reflection of this commitment.

CAV 2020 received a very high number of submissions (240). We accepted 18 tool papers, 4 case studies, and 43 regular papers, which amounts to an acceptance rate of roughly 27%. The accepted papers cover a wide spectrum of topics, from theoretical results to applications of formal methods. These papers apply or extend formal methods to a wide range of domains such as concurrency, machine learning, and industrially deployed systems. The program featured invited talks by David Dill (Calibra) and Pushmeet Kohli (Google DeepMind) as well as invited tutorials by Tevfik Bultan (University of California, Santa Barbara) and Sriram Sankaranarayanan (University of Colorado at Boulder). Furthermore, we continued the tradition of Logic Lounge, a series of discussions on computer science topics targeting a general audience.

In addition to the main conference, CAV 2020 hosted the following workshops: Numerical Software Verification (NSV), Verified Software: Theories, Tools, and Experiments (VSTTE), Verification of Neural Networks (VNN), Democratizing Software Verification, Synthesis (SYNT), Program Equivalence and Relational Reasoning (PERR), Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), Formal Methods for Blockchains (FMBC), and Verification Mentoring Workshop (VMW).

Organizing a flagship conference like CAV requires a great deal of effort from the community. The Program Committee (PC) for CAV 2020 consisted of 85 members – a committee of this size ensures that each member has to review a reasonable number of papers in the allotted time. In all, the committee members wrote over 960 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2020 PC for their outstanding efforts in evaluating the submissions and making sure that each paper got a fair chance. Like last year's CAV, we made the artifact evaluation mandatory for tool paper submissions and optional but encouraged for the rest of the accepted papers. The Artifact Evaluation Committee consisted of 40 reviewers who put in significant effort to evaluate each artifact. The goal of this process was to provide constructive feedback to tool developers and help make the research published in CAV more reproducible. The Artifact

Evaluation Committee was generally quite impressed by the quality of the artifacts, and, in fact, all accepted tools passed the artifact evaluation. Among the accepted regular papers, 67% of the authors submitted an artifact, and 76% of these artifacts passed the evaluation. We are also very grateful to the Artifact Evaluation Committee for their hard work and dedication in evaluating the submitted artifacts. The evaluation and selection process involved thorough online PC discussions using the EasyChair conference management system, resulting in more than 2,000 comments.

CAV 2020 would not have been possible without the tremendous help we received from several individuals, and we would like to thank everyone who helped make CAV 2020 a success. First, we would like to thank Xinyu Wang and He Zhu for chairing the Artifact Evaluation Committee and Jyotirmoy Deshmukh for local arrangements. We also thank Zvonimir Rakamaric for chairing the workshop organization, Clark Barrett for managing sponsorship, Thomas Wies for arranging student fellowships, and Yakir Vizel for handling publicity. We also thank Roopsha Samanta for chairing the Mentoring Committee. Last but not least, we would like to thank members of the CAV Steering Committee (Kenneth McMillan, Aarti Gupta, Orna Grumberg, and Daniel Kroening) for helping us with several important aspects of organizing CAV 2020.

We hope that you will find the proceedings of CAV 2020 scientifically interesting and thought-provoking!

June 2020

Shuvendu K. Lahiri
Chao Wang

Organization

Program Chairs

Shuvendu K. Lahiri	Microsoft Research, USA
Chao Wang	University of Southern California, USA

Workshop Chair

Zvonimir Rakamaric	University of Utah, USA

Sponsorship Chair

Clark Barrett	Stanford University, USA

Publicity Chair

Yakir Vizel	Technion - Israel Institute of Technology, Israel

Fellowship Chair

Thomas Wies	New York University, USA

Local Arrangements Chair

Jyotirmoy Deshmukh	University of Southern California, USA

Program Committee

Aws Albarghouthi	University of Wisconsin-Madison, USA
Jade Alglave	University College London, UK
Christel Baier	Technical University of Dresden, Germany
Gogul Balakrishnan	Google, USA
Sorav Bansal	India Institute of Technology, Delhi, India
Gilles Barthe	Max Planck Institute, Germany
Josh Berdine	Facebook, UK
Per Bjesse	Synopsys, USA
Sam Blackshear	Calibra, USA
Roderick Bloem	Graz University of Technology, Austria
Borzoo Bonakdarpour	Iowa State University, USA
Ahmed Bouajjani	Paris Diderot University, France
Tevfik Bultan	University of California, Santa Barbara, USA
Pavol Cerny	Vienna University of Technology, Austria

Sagar Chaki	Mentor Graphics, USA
Swarat Chaudhuri	University of Texas, Austin, USA
Hana Chockler	King's College London, UK
Maria Christakis	Max Planck Institute, Germany
Eva Darulova	Max Planck Institute, Germany
Cristina David	University of Cambridge, UK
Ankush Desai	Amazon, USA
Jyotirmoy Deshmukh	University of Southern California, USA
Cezara Dragoi	Inria, France
Kerstin Eder	University of Bristol, UK
Michael Emmi	Amazon, USA
Constantin Enea	Université de Paris, France
Lu Feng	University of Virginia, USA
Yu Feng	University of California, Santa Barbara, USA
Bernd Finkbeiner	Saarland University, Germany
Dana Fisman	Ben-Gurion University, Israel
Daniel J. Fremont	University of California, Santa Cruz, USA
Malay Ganai	Synopsys, USA
Ganesh Gopalakrishnan	University of Utah, USA
Orna Grumberg	Technion - Israel Institute of Technology, Israel
Arie Gurfinkel	University of Waterloo, Canada
Alan J. Hu	The University of British Columbia, Canada
Laura Humphrey	Air Force Research Laboratory, USA
Franjo Ivancic	Google, USA
Joxan Jaffar	National University of Singapore, Singapore
Dejan Jovanović	SRI International, USA
Zachary Kincaid	Princeton University, USA
Laura Kovacs	Vienna University of Technology, Austria
Daniel Kroening	University of Oxford, UK
Ori Lahav	Tel Aviv University, Israel
Akash Lal	Microsoft, India
Anthony Lin	TU Kaiserslautern, Germany
Yang Liu	Nanyang Technological University, Singapore
Francesco Logozzo	Facebook, USA
Ruben Martins	Carnegie Mellon University, USA
Anastasia Mavridou	NASA Ames Research Center, USA
Jedidiah McClurg	Colorado School of Mines, USA
Kenneth McMillan	Microsoft, USA
Kuldeep S. Meel	National University of Singapore, Singapore
Sayan Mitra	University of Illinois at Urbana-Champaign, USA
Ruzica Piskac	Yale University, USA
Xiaokang Qiu	Purdue University, USA
Mukund Raghothaman	University of Southern California, USA
Jan Reineke	Saarland University, Germany
Kristin Yvonne Rozier	Iowa State University, USA
Philipp Ruemmer	Uppsala University, Sweden

Krishna S	India Institute of Technology, Bombay, India
Sriram Sankaranarayanan	University of Colorado at Boulder, USA
Natarajan Shankar	SRI International, USA
Natasha Sharygina	University of Lugano, Switzerland
Sharon Shoham	Tel Aviv University, Israel
Alexandra Silva	University College London, UK
Anna Slobodova	Centaur Technology, USA
Fabio Somenzi	University of Colorado at Boulder, USA
Fu Song	ShanghaiTech University, China
Aditya Thakur	University of California, Davis, USA
Ashish Tiwari	Microsoft, USA
Aaron Tomb	Galois, Inc., USA
Ashutosh Trivedi	University of Colorado at Boulder, USA
Caterina Urban	Inria, France
Niki Vazou	IMDEA, Spain
Margus Veanes	Microsoft, USA
Yakir Vizel	Technion - Israel Institute of Technology, Israel
Xinyu Wang	University of Michigan, USA
Georg Weissenbacher	Vienna University of Technology, Austria
Fei Xie	Portland State University, USA
Jin Yang	Intel, USA
Naijun Zhan	Chinese Academy of Sciences, China
He Zhu	Rutgers University, USA

Artifact Evaluation Committee

Xinyu Wang (Co-chair)	University of Michigan, USA
He Zhu (Co-chair)	Rutgers University, USA
Angello Astorga	University of Illinois at Urbana-Champaign, USA
Subarno Banerjee	University of Michigan, USA
Martin Blicha	University of Lugano, Switzerland
Brandon Bohrer	Carnegie Mellon University, USA
Jose Cambronero	Massachusetts Institute of Technology, USA
Joonwon Choi	Massachusetts Institute of Technology, USA
Norine Coenen	Saarland University, Germany
Katherine Cordwell	Carnegie Mellon University, USA
Chuchu Fan	Massachusetts Institute of Technology, USA
Yotam Feldman	Tel Aviv University, Israel
Timon Gehr	ETH Zurich, Switzerland
Aman Goel	University of Michigan, USA
Chih-Duo Hong	University of Oxford, UK
Bo-Yuan Huang	Princeton University, USA
Jeevana Priya Inala	Massachusetts Institute of Technology, USA
Samuel Kaufman	University of Washington, USA
Ratan Lal	Kansas State University, USA
Stella Lau	Massachusetts Institute of Technology, USA

Juneyoung Lee	Seoul National University, South Korea
Enrico Magnago	Fondazione Bruno Kessler, Italy
Umang Mathur	University of Illinois at Urbana-Champaign, USA
Jedidiah McClurg	Colorado School of Mines, USA
Sam Merten	Ohio University, USA
Luan Nguyen	University of Pennsylvania, USA
Aina Niemetz	Stanford University, USA
Shankara Pailoor	The University of Texas at Austin, USA
Brandon Paulsen	University of Southern California, USA
Mouhammad Sakr	Saarland University, Germany
Daniel Selsam	Microsoft Research, USA
Jiasi Shen	Massachusetts Institute of Technology, USA
Xujie Si	University of Pennsylvania, USA
Gagandeep Singh	ETH Zurich, Switzerland
Abhinav Verma	Rice University, USA
Di Wang	Carnegie Mellon University, USA
Yuepeng Wang	The University of Texas at Austin, USA
Guannan Wei	Purdue University, USA
Zikang Xiong	Purdue University, USA
Klaus von Gleissenthall	University of California, San Diego, USA

Mentoring Workshop Chair

Roopsha Samanta	Purdue University, USA

Steering Committee

Kenneth McMillan	Microsoft Research, USA
Aarti Gupta	Princeton University, USA
Orna Grumberg	Technion - Israel Institute of Technology, Israel
Daniel Kroening	University of Oxford, UK

Additional Reviewers

Shaull Almagor
Sepideh Asadi
Angello Astorga
Brandon Bohrer
Vincent Cheval
Javier Esparza
Marie Farrell
Grigory Fedyukovich
Jerome Feret
James Hamil
Antti Hyvarinen
Matteo Marescotti
Rodrigo Ottoni
Junkil Park
Sean Regisford
David Sanan
Aritra Sengupta
Sadegh Soudjani
Tim Zakian

Contents – Part I

Contents – Part II

AI Verification

NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems

Hoang-Dung Tran[1,2], Xiaodong Yang[1], Diego Manzanas Lopez[1], Patrick Musau[1], Luan Viet Nguyen[3], Weiming Xiang[5], Stanley Bak[4], and Taylor T. Johnson[1](✉)

[1] University of Nebraska, Lincoln, USA
taylor.johnson@vanderbilt.edu
[2] Vanderbilt University, Nashville, USA
[3] University of Dayton, Dayton, USA
[4] Stony Brook University, Stony Brook, USA
[5] Augusta University, Augusta, USA

Abstract. This paper presents the Neural Network Verification (NNV) software tool, a set-based verification framework for deep neural networks (DNNs) and learning-enabled cyber-physical systems (CPS). The crux of NNV is a collection of reachability algorithms that make use of a variety of set representations, such as polyhedra, star sets, zonotopes, and abstract-domain representations. NNV supports both exact (sound and complete) and over-approximate (sound) reachability algorithms for verifying safety and robustness properties of feed-forward neural networks (FFNNs) with various activation functions. For learning-enabled CPS, such as closed-loop control systems incorporating neural networks, NNV provides exact and over-approximate reachability analysis schemes for linear plant models and FFNN controllers with piecewise-linear activation functions, such as ReLUs. For similar neural network control systems (NNCS) that instead have nonlinear plant models, NNV supports over-approximate analysis by combining the star set analysis used for FFNN controllers with zonotope-based analysis for nonlinear plant dynamics building on CORA. We evaluate NNV using two real-world case studies: the first is safety verification of ACAS Xu networks, and the second deals with the safety verification of a deep learning-based adaptive cruise control system.

The material presented in this paper is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089, the National Science Foundation (NSF) under grant numbers SHF 1910017 and FMitF 1918450, and the Air Force Office of Scientific Research (AFOSR) through award numbers FA9550-18-1-0122 and FA9550-19-1-0288. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 3–17, 2020.
https://doi.org/10.1007/978-3-030-53288-8_1

Keywords: Neural networks · Machine learning · Cyber-physical systems · Verification · Autonomy

1 Introduction

Deep neural networks (DNNs) have quickly become one of the most widely used tools for dealing with complex and challenging problems in numerous domains, such as image classification [10,16,25], function approximation, and natural language translation [11,18]. Recently, DNNs have been used in safety-critical cyber-physical systems (CPS), such as autonomous vehicles [8,9,52] and air traffic collision avoidance systems [21]. Although utilizing DNNs in safety-critical applications can demonstrate considerable performance benefits, assuring the safety and robustness of these systems is challenging because DNNs possess complex non-linear characteristics. Moreover, it has been demonstrated that their behavior can be unpredictable due to slight perturbations in their inputs (i.e., adversarial perturbations) [36].

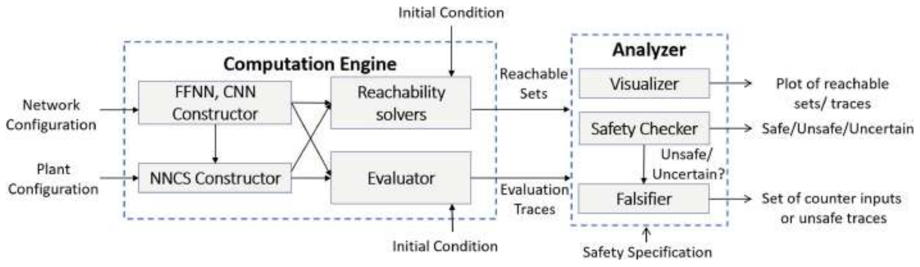


Fig. 1. An overview of NNV and its major modules and components.

In this paper, we introduce the NNV (**N**eural **N**etwork **V**erification) tool, which is a software framework that performs set-based verification for DNNs and learning-enabled CPS, known colloquially as neural network control systems (NNCS) as shown in Fig. 2[1]. NNV provides a set of reachability algorithms that can compute both the exact and over-approximate reachable sets of DNNs and NNCSs using a variety of set representations such as polyhedra [40,53–56], star sets [29,38,39,41], zonotopes [32], and abstract domain representations [33]. The reachable set obtained from NNV contains all possible states of a DNN from bounded input sets or of a NNCS from sets of initial states of a plant model. NNV declares a DNN or a NNCS to be safe if, and only if, their reachable sets do not violate safety properties (i.e., have a non-empty intersection with any state satisfying the negation of the safety property). If a safety property is violated,

[1] The source code for NNV is publicly available: https://github.com/verivital/nnv/. A CodeOcean capsule [43] is also available: https://doi.org/10.24433/CO.0221760.v1.

Table 1. Overview of major features available in NNV. Links refer to relevant files/-classes in the NNV codebase. BN refers to batch normalization layers, FC to fully-connected layers, AvgPool to average pooling layers, Conv to convolutional layers, and MaxPool to max pooling layers.

Feature	Exact analysis	Over-approximate analysis
Components	FFNN, CNN, NNCS	FFNN, CNN, NNCS
Plant dynamics (for NNCS)	Linear ODE	Linear ODE, Nonlinear ODE
Discrete/Continuous (for NNCS)	Discrete Time	Discrete Time, Continuous Time
Activation functions	ReLU, Satlin	ReLU, Satlin, Sigmoid, Tanh
CNN Layers	MaxPool, Conv, BN, AvgPool, FC	MaxPool, Conv, BN, AvgPool, FC
Reachability methods	Star, Polyhedron, ImageStar	Star, Zonotope, Abstract-domain, ImageStar
Reachable set/Flow-pipe Visualization	Yes	Yes
Parallel computing	Yes	Partially supported
Safety verification	Yes	Yes
Falsification	Yes	Yes
Robustness verification (for FFNN/CNN)	Yes	Yes
Counterexample generation	Yes	Yes

NNV can construct a complete set of counter-examples demonstrating the set of all possible unsafe initial inputs and states by using the star-based exact reachability algorithm [38,41]. To speed up computation, NNV uses parallel computing, as the majority of the reachability algorithms in NNV are more efficient when executed on multi-core platforms and clusters.

NNV has been successfully applied to safety verification and robustness analysis of several real-world DNNs, primarily feedforward neural networks (FFNNs) and convolutional neural networks (CNNs), as well as learning-enabled CPS. To highlight NNV's capabilities, we present brief experimental results from two case studies. The first compares methods for safety verification of the ACAS Xu networks [21], and the second presents safety verification of a learning-based adaptive cruise control (ACC) system.

2 Overview and Features

NNV is an object-oriented toolbox written in Matlab, which was chosen in part due to the prevalence of Matlab/Simulink in the design of CPS. NNV uses the MPT toolbox [26] for polytope-based reachability analysis and visualization [40], and makes use of CORA [3] for zonotope-based reachability analysis of nonlinear plant models [38]. NNV also utilizes the Neural Network Model Transformation Tool (NNMT) for transforming neural network models from Keras and Tensorflow into Matlab using the Open Neural Network Exchange (ONNX) format, and the Hybrid Systems Model Transformation and Translation tool (HyST) [5]

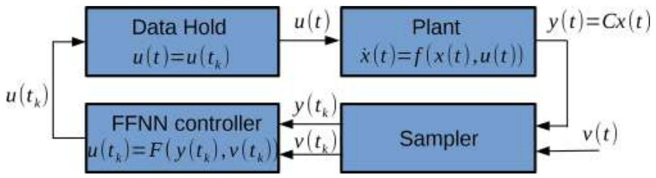


Fig. 2. Architecture of a typical neural network control system (NNCS).

for plant configuration. NNV makes use of YALMIP [27] for some optimization problems and MatConvNet [46] for some CNN operations.

The NNV toolbox contains two main modules: a *computation engine* and an *analyzer*, shown in Fig. 1. The computation engine module consists of four subcomponents: 1) the *FFNN constructor*, 2) the *NNCS constructor*, 3) *the reachability solvers*, and 4) *the evaluator*. The FFNN constructor takes a network configuration file as an input and generates a FFNN object. The NNCS constructor takes the FFNN object and the plant configuration, which describes the dynamics of a system, as inputs and then creates an NNCS object. Depending on the application, either the FFNN (or NNCS) object will be fed into a reachability solver to compute the reachable set of the FFNN (or NNCS) from a given initial set of states. Then, the obtained reachable set will be passed to the analyzer module. The analyzer module consists of three subcomponents: 1) a *visualizer*, 2) a *safety checker*, and 3) a *falsifier*. The visualizer can be called to plot the obtained reachable set. Given a safety specification, the safety checker can reason about the safety of the FFNN or NNCS with respect to the specification. When an exact (sound and complete) reachability solver is used, such as the star-based solver, the safety checker can return either "safe," or "unsafe" along with a set of counterexamples. When an over-approximate (sound) reachability solver is used, such as the zonotope-based scheme or the approximate star-based solvers, the safety checker can return either "safe" or "*uncertain*" (unknown). In this case, the falsifier automatically calls the evaluator to generate simulation traces to find a counterexample. If the falsifier can find a counterexample, then NNV returns unsafe. Otherwise, it returns unknown. Table 1 shows a summary of the major features of NNV.

3 Set Representations and Reachability Algorithms

NNV implements a set of reachability algorithms for *sequential* FFNNs and CNNs, as well as NNCS with FFNN controllers as shown in Fig. 2. The reachable set of a sequential FFNN is computed layer-by-layer. The output reachable set of a layer is the input set of the next layer in the network.

3.1 Polyhedron [40]

The polyhedron reachability algorithm computes the exact polyhedron reachable set of a FFNN with ReLU activation functions. The exact reachability computation of layer L in a FFNN is done as follows. First, we construct the affine mapping $\bar{I}$ of the input polyhedron set I, using the weight matrix W and the bias vector b, i.e., $\bar{I} = W \times I + b$. Then, the exact reachable set of the layer R_L is constructed by executing a sequence of stepReLU operations, i.e., $R_L = stepReLU_n(stepReLU_{n-1}(\cdots(stepReLU_1(\bar{I}))))$. Since a *stepReLU* operation can split a polyhedron into two new polyhedra, the exact reachable set of a layer in a FFNN is usually a union of polyhedra. The polyhedron reachability algorithm is computationally expensive because computing affine mappings with polyhedra is costly. Additionally, when computing the reachable set, the polyhedron approach extensively uses the expensive conversion between the H-representation and the V-representation. These are the main drawbacks that limit the scalability of the polyhedron approach. Despite that, we extend the polyhedron reachability algorithm for NNCSs with FFNN controllers. However, the propagation of polyhedra in NNCS may lead to a large degree of conservativeness in the computed reachable set [38].

3.2 Star Set [38,41] (code)

The star set is an efficient set representation for simulation-based verification of large linear systems [6,7,42] where the superposition property of a linear system can be exploited in the analysis. It has been shown in [41] that the star set is also suitable for reachability analysis of FFNNs. In contrast to polyhedra, the affine mapping and intersection with a half space of a star set is more easily computed. NNV implements an enhanced version of the exact and over-approximate reachability algorithms for FFNNs proposed in [41] by minimizing the number of LP optimization problems that need to be solved in the computation. The exact algorithm that makes use of star sets is similar to the polyhedron method that makes use of *stepReLU* operations. However, it is much faster and more scalable than the polyhedron method because of the advantage that star sets have in affine mapping and intersection. The approximate algorithm obtains an over-approximation of the exact reachable set by approximating the exact reachable set after applying an activation function, e.g., ReLU, Tanh, Sigmoid. We refer readers to [41] for a detailed discussion of star-set reachability algorithms for FFNNs.

We note that NNV implements enhanced versions of earlier star-based reachability algorithms [41]. Particularly, we minimize the number of linear programming (LP) optimization problems that must be solved in order to construct the reachable set of a FFNN by quickly estimating the ranges of all of the states in the star set using only the ranges of the predicate variables. Additionally, the extensions of the star reachability algorithms to NNCS with linear plant models can eliminate the explosion of conservativeness in the polyhedron method [38,39]. The reason behind this is that in star sets, the relationship between the plant state variables and the control inputs is preserved in the computation since they are defined by a unique set of predicate variables. We refer readers to [38,39] for a detailed discussion of the extensions of the star-based reachability algorithms for NNCSs with linear/nonlinear plant models.

3.3 Zonotope [32] (code)

NNV implements the zonotope reachability algorithms proposed in [32] for FFNNs. Similar to the over-approximate algorithm using star sets, the zonotope algorithm computes an over-approximation of the exact reachable set of a FFNN. Although the zonotope reachability algorithm is very fast and scalable, it produces a very conservative reachable set in comparison to the star set method as shown in [41]. Consequently, zonotope-based reachability algorithms are usually only more efficient for very small input sets. As an example it can be more suitable for robustness certification.

3.4 Abstract Domain [33]

NNV implements the abstract domain reachability algorithm proposed in [33] for FFNNs. NNV's abstract domain reachability algorithm specifies an abstract domain as a star set and estimates the *over-approximate ranges* of the states based on the ranges of the new introduced predicate variables. We note that better ranges of the states can be computed by solving LP optimization. However, better ranges come with more computation time.

3.5 ImageStar Set [37] (code)

NNV recently introduced a new set representation called the ImageStar for use in the verification of deep convolutional neural networks (CNNs). Briefly, the ImageStar is a generalization of the star set where the anchor and generator vectors are replaced by multi-channel images. The ImageStar is efficient in the analysis of convolutional layers, average pooling layers, and fully connected layers, whereas max pooling layers and ReLU layers consume most of the computation time. NNV implements exact and over-approximate reachability algorithms using the ImageStar for serial CNNs. In short, using the ImageStar, we can analyze the robustness under adversarial attacks of the real-world VGG16 and VGG19 deep perception networks [31] that consist of >100 million parameters [37].

4 Evaluation

The experiments presented in this section were performed on a desktop with the following configuration: Intel Core i7-6700 CPU @ 3.4 GHz 8 core Processor, 64 GB Memory, and 64-bit Ubuntu 16.04.3 LTS OS.

4.1 Safety Verification of ACAS Xu Networks

We evaluate NNV in comparison to Reluplex [22], Marabou [23], and ReluVal [49], by considering the verification of safety property ϕ_3 and ϕ_4 of the ACAS Xu neural networks [21] for all 45 networks.[2] All the experiments were done using 4 cores for computation. The results are summarized in Table 2 where (SAT) denotes the networks are safe, (UNSAT) is unsafe, and (UNK) is unknown. We note that (UNK) may occur due to the conservativeness of the reachability analysis scheme. Detailed verification results are presented in the appendix of the extended version of this paper [44]. For a fast comparison with other tools, we also tested a subset of the inputs for Property 1–4 on all the 45 networks. We note that the polyhedron method [40] achieves a timeout on most of networks, and therefore, we neglect this method in the comparison.

Verification Time. For property ϕ_3, NNV's exact-star method is about 20.7$\times$ faster than Reluplex, 14.2$\times$ faster than Marabou, 81.6$\times$ faster than Marabou-DnC (i.e., divide and conquer method). The approximate star method is 547$\times$ faster than Reluplex, 374$\times$ faster than Marabou, 2151$\times$ faster than Marabou-DnC, and 8$\times$ faster than ReluVal. For property ϕ_4, NNV's exact-star method is 25.3$\times$ faster than Reluplex, 18.0$\times$ faster than Marabou, 53.4$\times$ faster than Marabou-DnC, while the approximate star method is 625$\times$ faster than Reluplex, 445$\times$ faster than Marabou, 1321$\times$ faster than Marabou-DnC.

Table 2. Verification results of ACAS Xu networks.

ACAS XU ϕ_3	SAT	UNSAT	UNK	TIMEOUT			TIME(s)
				1 h	2 h	10 h	
Reluplex	3	42	0	2	0	0	28454
Marabou	3	42	0	1	0	0	19466
Marabou DnC	3	42	0	3	3	1	111880
ReluVal	3	42	0	0	0	0	416
Zonotope	0	2	43	0	0	0	3
Abstract Domain	0	0	45	0	0	0	8
NNV Exact Star	3	42	0	0	0	0	1371
NNV Appr. Star	0	29	16	0	0	0	52
ACAS XU ϕ_4							
Reluplex	3	42	0	0	0	0	11880
Marabou	3	42	0	0	0	0	8470
Marabou DnC	3	42	0	2	2	0	25110
ReluVal	3	42	0	0	0	0	27
Zonotope	0	1	44	0	0	0	5
Abstract Domain	0	0	45	0	0	0	7
NNV Exact Star	3	42	0	0	0	0	470
NNV Appr. Star	0	32	13	0	0	0	19

[2] We omit properties ϕ_1 and ϕ_2 for space and due to their long runtimes, but they can be reproduced in the artifact.

Conservativeness. The approximate star method is much less conservative than the zonotope and abstract domain methods. This is illustrated since it can verify more networks than the zonotope and abstract domain methods, and is because it obtains a tighter over-approximate reachable set. For property ϕ_3, the zonotope and abstract domain methods can prove safety of 2/45 networks, (4.44%) and 0/45 networks, (0%) respectively, while NNV's approximate star method can prove safety of 29/45 networks, (64.4%). For property ϕ_4, the zonotope and abstract domain method can prove safety of 1/45 networks, (2.22%) and 0/45 networks, (0.00%) respectively while the approximate star method can prove safety of 32/45, (71.11%).

4.2 Safety Verification of Adaptive Cruise Control System

To illustrate how NNV can be used to verify/falsify safety properties of learning-enabled CPS, we analyze a learning-based ACC system [1,38], in which the ego (following) vehicle has a radar sensor to measure the distance to the lead vehicle in the same lane, D_{rel}, as well as the relative velocity of the lead vehicle, V_{rel}. The ego vehicle has two control modes. In speed control mode, it travels at a driver-specified set speed $V_{set} = 30$, and in spacing control mode, it maintains a safe distance from the lead vehicle, D_{safe}. We train a neural network with 5 layers of 20 neurons per layer with ReLU activation functions to control the ego vehicle using a control period of 0.1 s.

We investigate safety of the learning-based ACC system with two types of plant dynamics: 1) a discrete linear plant, and 2) a nonlinear continuous plant governed by the following differential equations:

$$\begin{aligned}
&\dot{x}_{lead}(t) = v_{lead}(t),\ \dot{v}_{lead}(t) = \gamma_{lead}, && \dot{\gamma}_{lead}(t) = -2\gamma_{lead}(t) + 2a_{lead} - \mu v_{lead}^2(t),\\
&\dot{x}_{ego}(t) = v_{ego}(t),\ \dot{v}_{ego}(t) = \gamma_{ego}, && \dot{\gamma}_{ego}(t) = -2\gamma_{ego}(t) + 2a_{ego} - \mu v_{ego}^2(t),
\end{aligned}$$

where $x_{lead}(x_{ego})$, $v_{lead}(v_{ego})$ and $\gamma_{lead}(\gamma_{ego})$ are the position, velocity and acceleration of the lead (ego) vehicle respectively. $a_{lead}(a_{ego})$ is the acceleration control input applied to the lead (ego) vehicle, and $\mu = 0.0001$ is a friction parameter. To obtain a discrete linear model of the plant, we let $\mu = 0$ and discretize the corresponding linear continuous model using a zero-order hold on the inputs with a sample time of 0.1 s (i.e., the control period).

Verification Problem. The scenario we are interested in is when the two vehicles are operating at a safe distance between them and the ego vehicle is in speed control mode. In this state the lead vehicle driver suddenly decelerates with $a_{lead} = -5$ to reduce the speed. We want to verify if the neural network controller on the ego vehicle will decelerate to maintain a safe distance between the two vehicles. To guarantee safety, we require that $D_{rel} = x_{lead} - x_{ego} \geq D_{safe} = D_{default} + T_{gap} \times v_{ego}$ where $T_{gap} = 1.4$ s and $D_{default} = 10$. Our analysis investigates whether the safety requirement holds during the 5 s after the lead vehicle decelerates. We consider safety of the system under the following initial conditions: $x_{lead}(0) \in [90, 92]$, $v_{lead}(0) \in [20, 30]$, $\gamma_{lead}(0) = \gamma_{ego}(0) = 0$, $v_{ego}(0) \in [30, 30.5]$, and $x_{ego} \in [30, 31]$.

Table 3. Verification results for ACC system with different plant models, where VT is the verification time (in seconds).

v_lead(0)	Linear plant		Nonlinear plant	
	$Safety$	$VT(s)$	$Safety$	$VT(s)$
[29, 30]	SAFE	9.60	UNSAFE	346.62
[28, 29]	SAFE	9.45	UNSAFE	277.50
[27, 28]	SAFE	9.82	UNSAFE	289.70
[26, 27]	UNSAFE	17.80	UNSAFE	315.60
[25, 26]	UNSAFE	19.24	UNSAFE	305.56
[24, 25]	UNSAFE	18.12	UNSAFE	372.00

Verification Results. For linear dynamics, NNV can compute both the exact and over-approximate reachable sets of the ACC system in bounded time steps, while for nonlinear dynamics, NNV constructs an over-approximation of the reachable sets. The verification results for linear and nonlinear models using the over-approximate star method are presented in Table 3, which shows that safety of the ACC system depends on the initial velocity of the lead vehicle. When the initial velocity of the lead vehicle is smaller than 27 (m/s), the ACC system with the discrete plant model is unsafe. Using the exact star method, NNV can construct a *complete* set of counter-example inputs. When the over-approximate star method is used, if there is a potential safety violation, NNV simulates the system with 1000 random inputs from the input set to find counter examples. If a counterexample is found, the system is *UNSAFE*, otherwise, NNV returns a safety result of *UNKNOWN*. Figure 3 visualizes the reachable sets of the relative distance D_{rel} between two vehicles versus the required safe distance D_{safe} over time for two cases of initial velocities of the lead vehicle: $v_{lead}(0) \in [29, 30]$ and $v_{lead}(0) \in [24, 25]$. We can see that in the first case, $D_{ref} \geq D_{safe}$ for all 50 time steps stating that the system is safe. In the second case, $D_{ref} < D_{safe}$ in some control steps, so the system is unsafe. NNV supports a *reachLive* method to perform analysis and reachable set visualization on-the-fly to help the user observe the behavior of the system during verification.

The verification results for the ACC system with the nonlinear model are all *UNSAFE*, which is surprising. Since the neural network controller of the ACC system was trained with the linear model, it works quite well for the linear model. However, when a small friction term is added to the linear model to form a nonlinear model, the neural network controller's performance, in terms of safety, is significantly reduced. This problem raises an important issue in training neural network controllers using simulation data, and these schemes may not work in real systems since there is always a mismatch between the plant model in the simulation engine and the real system.

Verification Times. As shown in Table 3, the approximate analysis of the ACC system with discrete linear plant model is fast and can be done in 84 s. NNV

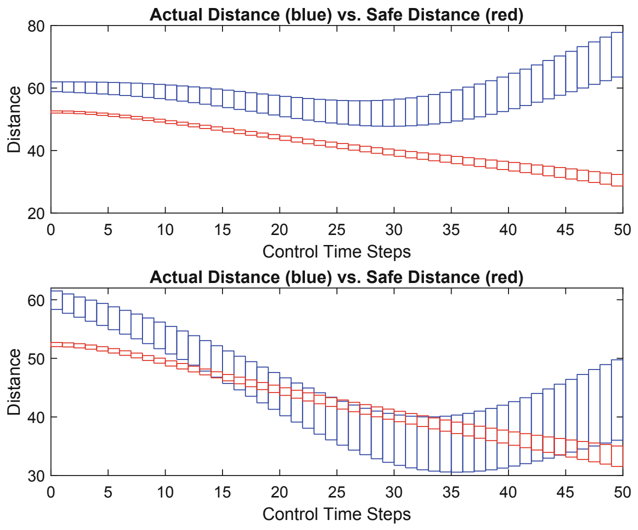


Fig. 3. Two scenarios of the ACC system. In the first (top) scenario ($v_{lead}(0) \in [29, 30]$ m/s), safety is guaranteed, $D_{rel} \geq D_{safe}$. In the second scenario (bottom) ($v_{lead}(0) \in [24,\ 25]$ m/s), safety is violated since $D_{ref} < D_{safe}$ in some control steps.

also supports exact analysis, but is computationally expensive as it constructs all reachable states. Because there are splits in the reachable sets of the neural network controller, the number of star sets in the reachable set of the plant increases quickly over time [38]. In contrast, the over-approximate method computes the interval hull of all reachable sets at each time step, and maintains a single reachable set of the plant throughout the computation. This makes the over-approximate method faster than the exact method. In terms of plant models, the nonlinear model requires more computation time than the linear one. As shown in Table 3, the verification for the linear model using the over-approximate method is 22.7× faster on average than of the nonlinear model.

5 Related Work

NNV was inspired by recent work in the emerging fields of neural network and machine learning verification. For the "open-loop" verification problem (verification of DNNs), many efficient techniques have been proposed, such as SMT-based methods [22,23,30], mixed-integer linear programming methods [14,24,28], set-based methods [4,17,32,33,48,50,53,57], and optimization methods [51,58]. For the "closed-loop" verification problem (NCCS verification), we note that the Verisig approach [20] is efficient for NNCS with nonlinear plants and with Sigmoid and Tanh activation functions. Additionally, the recent regressive polynomial rule inference approach [34] is efficient for safety verification of NNCS with nonlinear plant models and ReLU activation functions. The satisfiability modulo convex (SMC) approach [35] is also promising for NNCS with discrete linear plants, as it provides both soundness and completeness guarantees. ReachNN [19] is a recent approach that can efficiently control the conservativeness in the reachability analysis of NNCS with nonlinear plants and ReLU, Sigmoid, and Tanh activation functions in the controller. In [54], a novel simulation-guided approach has been developed to reduce significantly the computation cost for verification of NNCS. In other learning-enabled systems, falsification and testing-based approaches [12,13,45] have shown a significant promise in enhancing the safety of systems where perception components and neural networks interact with the physical world. Finally, there is significant related work in the domain of safe reinforcement learning [2,15,47,59], and combining guarantees from NNV with those provided in these methods would be interesting to explore.

6 Conclusions

We presented NNV, a software tool for the verification of DNNs and learning-enabled CPS. NNV provides a collection of reachability algorithms that can be used to verify safety (and robustness) of real-world DNNs, as well as learning-enabled CPS, such as the ACC case study. For closed-loop systems, NNV can compute the exact and over-approximate reachable sets of a NNCS with linear plant models. For NNCS with nonlinear plants, NNV computes an over-approximate reachable set and uses it to verify safety, but can also automatically falsify the system to find counterexamples.

References

1. Model Predictive Control Toolbox. The MathWorks Inc., Natick, Massachusetts (2019). https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
3. Althoff, M.: An introduction to cora 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
4. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 731–744. Association for Computing Machinery, New York (2019)
5. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 128–133. ACM (2015)
6. Bak, S., Duggirala, P.S.: Simulation-equivalent reachability of large linear systems with inputs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 401–420. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_20

7. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 23–32. ACM (2019)
8. Bojarski, M., et al.: End to end learning for self-driving cars (2016). arXiv preprint arXiv:1604.07316
9. Chen, C., Seff, A., Kornhauser, A., Xiao, J.: Deepdriving: Learning affordance for direct perception in autonomous driving. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2722–2730 (2015)
10. Cireşan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification (2012). arXiv preprint arXiv:1202.2745
11. Collobert, R., Weston, J.: A unified architecture for natural language processing: Deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning, pp. 160–167. ACM (2008)
12. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: NASA Formal Methods Symposium, pp. 357–372. Springer (2017)
13. Dreossi, T., et al.: VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 432–442. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_25
14. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks (2017). arXiv preprint arXiv:1709.09130
15. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 413–430. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_28
16. Gatys, L.A., Ecker, A.S., Bethge, M.: Image style transfer using convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2414–2423 (2016)
17. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP) (2018)
18. Goldberg, Y.: A primer on neural network models for natural language processing. J. Artif. Intell. Res. **57**, 345–420 (2016)
19. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: Reachability analysis of neural-network controlled systems (2019). arXiv preprint arXiv:1906.10654
20. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Hybrid Systems: Computation and Control (HSCC) (2019)
21. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J.: Policy compression for aircraft collision avoidance systems. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), pp. 1–10. IEEE (2016)
22. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
23. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
24. Kouvaros, P., Lomuscio, A.: Formal verification of cnn-based perception systems (2018). arXiv preprint arXiv:1811.11373

25. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
26. Kvasnica, M., Grieder, P., Baotić, M., Morari, M.: Multi-parametric toolbox (MPT). In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 448–462. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_30
27. Löfberg, J.: Yalmip : A toolbox for modeling and optimization in MATLAB. In: Proceedings of the CACSD Conference,Taipei, Taiwan (2004). http://users.isy.liu.se/johanl/yalmip
28. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks (2017). arXiv preprint arXiv:1706.07351
29. Lopez, D.M., Musau, P., Tran, H.D., Johnson, T.T.: Verification of closed-loop systems with neural network controllers. In: Frehse, G., Althoff, M. (eds.) ARCH19, 6th International Workshop on Applied Verification of Continuous and Hybrid Systems, EPiC Series in Computing, vol. 61, pp. 201–210. EasyChair (2019)
30. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
31. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). arXiv preprint arXiv:1409.1556
32. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems, pp. 10825–10836 (2018)
33. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 1–30 (2019). Article 41
34. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Hybrid Systems: Computation and Control (HSCC) (2019)
35. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Hybrid Systems: Computation and Control (HSCC) (2019)
36. Szegedy, C., et al.: Intriguing properties of neural networks (2013). arXiv preprint arXiv:1312.6199
37. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In: 32nd International Conference on Computer-Aided Verification (CAV). Springer (2020)
38. Tran, H.D., Cei, F., Lopez, D.M., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. In: ACM SIGBED International Conference on Embedded Software (EMSOFT 2019). ACM (2019)
39. Tran, H.D., Cei, F., Lopez, D.M., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control (July 2019)
40. Tran, H.D., et al.: Parallelizable reachability analysis algorithms for feed-forward neural networks. In: 7th International Conference on Formal Methods in Software Engineering (FormaliSE2019), Montreal, Canada (2019)
41. Tran, H.D., et al.: Star-based reachability analysis for deep neural networks. In: 23rd International Symposium on Formal Methods, FM 2019. Springer International Publishing (2019)

42. Tran, H.D., Nguyen, L.V., Hamilton, N., Xiang, W., Johnson, T.T.: Reachability analysis for high-index linear differential algebraic equations (daes). In: 17th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2019). Springer International Publishing (2019)
43. Tran, H.D., et al.: NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems (CodeOcean Capsule) (2020). https://doi.org/10.24433/CO.0221760.v1
44. Tran, H.D., et al.: NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems (2020). arXiv preprint arXiv:2004.05519
45. Tuncali, C.E., Fainekos, G., Ito, H., Kapinski, J.: Simulation-based adversarial test generation for autonomous vehicles with machine learning components (2018). arXiv preprint arXiv:1804.06760
46. Vedaldi, A., Lenc, K.: Matconvnet: Convolutional neural networks for matlab. In: Proceedings of the 23rd ACM International Conference on Multimedia, pp. 689–692. ACM (2015)
47. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, PMLR, 10–15 Jul 2018, vol. 80, pp. 5045–5054 (2018)
48. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems, pp. 6369–6379 (2018)
49. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore (2018)
50. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals (2018). arXiv preprint arXiv:1804.10829
51. Weng, T.W., et al.: Towards fast computation of certified robustness for relu networks (2018). arXiv preprint arXiv:1804.09699
52. Wu, B., Iandola, F.N., Jin, P.H., Keutzer, K.: Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In: CVPR Workshops, pp. 446–454 (2017)
53. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)
54. Xiang, W., Tran, H.D., Yang, X., Johnson, T.T.: Reachable set estimation for neural network control systems: A simulation-guided approach. IEEE Trans. Neural Netw. Learn. Syst. 1–10 (2020)
55. Xiang, W., Tran, H.D., Johnson, T.T.: Reachable set computation and safety verification for neural networks with relu activations (2017). arXiv preprint arXiv:1712.08163
56. Xiang, W., Tran, H.D., Johnson, T.T.: Specification-guided safety verification for feedforward neural networks. In: AAAI Spring Symposium on Verification of Neural Networks (2019)
57. Yang, X., Tran, H.D., Xiang, W., Johnson, T.: Reachability analysis for feedforward neural networks using face lattices (2020). arXiv preprint arXiv:2003.01226

58. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems, pp. 4944–4953 (2018)
59. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 686–701. Association for Computing Machinery, New York (2019)

Verification of Deep Convolutional Neural Networks Using ImageStars

Hoang-Dung Tran[1,2], Stanley Bak[3], Weiming Xiang[4], and Taylor T. Johnson[2](✉)

[1] University of Nebraska, Lincoln, USA
[2] Vanderbilt University, Nashville, USA
taylor.johnson@vanderbilt.edu
[3] Stony Brook University, Stony Brook, USA
[4] Augusta University, Augusta, USA

Abstract. Convolutional Neural Networks (CNN) have redefined state-of-the-art in many real-world applications, such as facial recognition, image classification, human pose estimation, and semantic segmentation. Despite their success, CNNs are vulnerable to adversarial attacks, where slight changes to their inputs may lead to sharp changes in their output in even well-trained networks. Set-based analysis methods can detect or prove the absence of bounded adversarial attacks, which can then be used to evaluate the effectiveness of neural network training methodology. Unfortunately, existing verification approaches have limited scalability in terms of the size of networks that can be analyzed. In this paper, we describe a set-based framework that successfully deals with real-world CNNs, such as VGG16 and VGG19, that have high accuracy on ImageNet. Our approach is based on a new set representation called the ImageStar, which enables efficient exact and over-approximative analysis of CNNs. ImageStars perform efficient set-based analysis by combining operations on concrete images with linear programming (LP). Our approach is implemented in a tool called NNV, and can verify the robustness of VGG networks with respect to a small set of input states, derived from adversarial attacks, such as the DeepFool attack. The experimental results show that our approach is less conservative and faster than existing zonotope and polytope methods.

Keywords: Neural networks · Reachability analysis · Machine learning · Computer vision

The material presented in this paper is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089, the National Science Foundation (NSF) under grant numbers SHF 1910017 and FMitF 1918450, and the Air Force Office of Scientific Research (AFOSR) through award numbers FA9550-18-1-0122 and FA9550-19-1-0288. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR, DARPA, or NSF. An extended version of this paper with appendices is available [33].

1 Introduction

Convolutional neural networks (CNN) have rapidly accelerated progress in computer vision with many practical applications such as face recognition [19], image classification [18], document analysis [21] and semantic segmentation. Recently, it has been shown that CNNs are vulnerable to adversarial attacks, where a well-trained CNN can be fooled into producing errant predictions due to tiny changes in their inputs [9]. Many applications such as autonomous driving seek to leverage the power of CNNs. However due the opaque nature of these models there are reservations about using in safety-critical applications. Thus, there is an urgent need for formally evaluating the robustness of a trained CNN.

Formal verification of deep neural networks (DNNs) has recently become an important topic. The majority of existing approaches focus on verifying safety and robustness properties of feedforward neural networks (FNN) with the Rectified Linear Unit activation function (ReLU). These approaches include: mixed-integer linear programming (MILP) [5,17,23], satisfiability (SAT) and satisfiability modulo theory (SMT) techniques [7,15], optimization [6,11,22,42,44,51], and geometric reachability [29,30,36,37,41,43,45,47,48,50]. Adjacent to these methods are property inference techniques for DNNs, which are also an important and interesting research area being investigated [10]. In a similar fashion, the problem of verifying safety of cyber-physical systems (CPS) with learning-enabled neural network components with imperfect plant models and sensing information has recently attracted significant attention due to their real world applications [1,12–14,24,31,32,35,46,49]. This research area views the safety verification problem in a holistic manner by considering safety of the entire system where learning-enabled components interact with the physical world.

Although numerous tools and methods have been proposed for neural network verification, only a handful of methods can deal with CNNs [2,16,17,27, 29,30]. Moreover, in the aforementioned techniques, only one [27] can deal with real-world CNNs, such as VGGNet [28]. Their approach makes used of the concept of the L_0 distance between two images. Their optimization-based approach computes a tight bound on the number of pixels that may be changed in an image without affecting the classification result of the network. It can also efficiently generate adversarial examples that can be used to improve the robustness of network. In a similar manner, this paper seeks to verify robustness of real-world deep CNNs. Thus, we develop a set-based analysis method through the use of the *ImageStar*, a new set representation that can represent an infinite family of images. As an example, this representation can be used to represent a set of images distorted by an adversarial attack. Using the ImageStar, we develop both exact and over-approximate reachability algorithms to construct reachable sets that contain all the possible outputs of a CNN under an adversarial attack. These reachable sets are then used to reason about the overall robustness of the network. When a CNN violates a robustness property, our exact reachability scheme can construct a *set of concrete adversarial examples*. Our approach differs from [27] in two primary ways. First, our method does not provide robustness guarantees for a network in terms of the number of pixels that are allowed to be changed (in terms of L_0 distance). Instead, we prove the robustness of the network on images that are attacked by disturbances bounded by arbitrary linear constraints. Second, our approach relies on reachable set computation of a network corresponding to a bounded input set, as opposed to an optimization-based approach. We implement these methods in the NNV tool [39] and compare with the zonotope method used in DeepZ [29] and the polytope method used in DeepPoly [30]. The experimental results indicate our method is less conservative and faster than existing approaches when verifying robustness of CNNs.

The main contributions of the paper include the following. First is the ImageStar set representation, which is an efficient representation for reachability analysis of CNNs. Second are exact and over-approximate reachability algorithms for constructing reachable sets and verifying robustness of CNNs. Third is the implementation of the ImageStar representation and reachability algorithms in NNV [39]. Fourth is a rigorous evaluation and comparison of proposed approaches, such as zonotope and polytope methods on different CNNs.

2 Problem Formulation

The reachability problem for CNNs is the task of analyzing a trained CNN with respect to some perturbed input set in order to construct a set containing all possible outputs of the network. In this paper, we consider the reachability of a CNN $\mathcal{N}$ that consists of a series of layers L that may include convolutional layers, fully connected layers, max-pooling layers, average pooling layers, and ReLU activation layers. Mathematically, we define a CNN with n layers as $\mathcal{N} = \{L_i\}, i = 1, 2, \dots, n$. The reachability of the CNN $\mathcal{N}$ is defined based on the concept of *reachable sets.*

Definition 1 (Reachable set of a CNN). *An (output) reachable set $\mathcal{R}_\mathcal{N}$ of a CNN $\mathcal{N} = \{L_i\}, i = 1, 2, \dots, n$ corresponding to a linear input set $\mathcal{I}$ is defined incrementally as:*

$$\begin{aligned}
\mathcal{R}_{L_1} &\triangleq \{y_1 \mid y_1 = L_1(x),\ x \in \mathcal{I}\},\\
\mathcal{R}_{L_2} &\triangleq \{y_2 \mid y_2 = L_2(y_1),\ y_1 \in \mathcal{R}_{L_1}\},\\
&\vdots\\
\mathcal{R}_\mathcal{N} = \mathcal{R}_{L_n} &\triangleq \{y_n \mid y_n = L_{n-1}(y_{n-1}),\ y_{n-1} \in \mathcal{R}_{L_{n-1}}\},
\end{aligned}$$

where $L_i(\cdot)$ is a function representing the operation of the i^{th} layer.

The definition shows that the reachable set of the CNN $\mathcal{N}$ can be constructed *layer-by-layer.* The core computation is constructing the reachable set of each layer L_i defined by a specific operation, i.e., convolution, affine mapping, max pooling, average pooling, or ReLU.

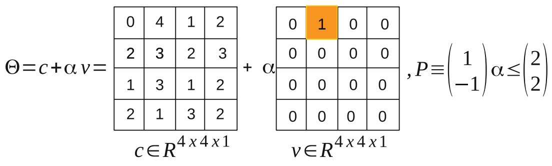


Fig. 1. An example of an ImageStar.

3 ImageStar

Definition 2. ***An ImageStar*** *Θ is a tuple $\langle c, V, P\rangle$ where $c \in \mathbb{R}^{h\times w\times nc}$ is the anchor image, $V = \{v_1, v_2, \cdots, v_m\}$ is a set of m images in $\mathbb{R}^{h\times w\times nc}$ called generator images, $P : \mathbb{R}^m \to \{\top, \bot\}$ is a predicate, and h, w, nc are the height, width, and number of channels of the images, respectively. The generator images are arranged to form the ImageStar's $h \times w \times nc \times m$ basis array. The set of images represented by the ImageStar is:*

$$[\![\Theta]\!] = \{x \mid x = c + \Sigma_{i=1}^{m}(\alpha_i v_i) \text{ such that } P(\alpha_1, \cdots, \alpha_m) = \top\}.$$

Sometimes we will refer to both the tuple Θ and the set of states $[\![\Theta]\!]$ as Θ. In this work, we restrict the predicates to be a conjunction of linear constraints, $P(\alpha) \triangleq C\alpha \leq d$ where, for p linear constraints, $C \in \mathbb{R}^{p\times m}$, α is the vector of m-variables, i.e., $\alpha = [\alpha_1, \cdots, \alpha_m]^T$, and $d \in \mathbb{R}^{p\times 1}$. A ImageStar is an empty set if and only if $P(\alpha)$ is empty.

Example 1 (ImageStar). A $4 \times 4 \times 1$ gray image with a bounded disturbance $b \in [-2, 2]$ applied on the pixel of the position $(1, 2, 1)$ can be described as an ImageStar depicted in Fig. 1.

Remark 1. An ImageStar is an extension of the generalized star set recently defined in [3,4,37,38]. In a generalized star set, the anchor and the generators are vectors, while in an ImageStar, the anchor and generators are images with multiple channels. We will later show that the ImageStar is a very efficient representation for the reachability analysis of convolutional layers, fully connected layers, and average pooling layers.

Proposition 1 (Affine mapping of an ImageStar). *An affine mapping of an ImageStar $\Theta = \langle c, V, P\rangle$ with a scale factor γ and an offset image β is another ImageStar $\Theta' = \langle c', V', P'\rangle$ in which the new anchor, generators and predicate are as follows:*

$$c' = \gamma \times c + \beta, \quad V' = \gamma \times V, \quad P' \equiv P.$$

Note that, the scale factor γ can be a scalar or a vector containing scalar scale factors in which each factor is used to scale one channel in the ImageStar.

4 Reachability of CNN Using ImageStars

In this section, we present the reachable set computation for the convolutional, average pooling, fully connected, batch normalization, max pooling, and ReLU layers with respect to an input set consisting of an ImageStar.

4.1 Reachability of a Convolutional Layer

We consider a two-dimensional convolutional layer with following parameters: the weights $W_{Conv2d} \in \mathbb{R}^{h_f \times w_f \times nc \times nf}$, the bias $b_{Conv2d} \in \mathbb{R}^{1 \times 1 \times nf}$, the padding size P, the stride S, and the dilation factor D where h_f, w_f, nc are the height, width, and the number of channels of the filters in the layer respectively. Additionally, nf is the number of filters. The reachability of a convolutional layer is given in the following lemma.

Lemma 1. *The reachable set of a convolutional layer with an ImageStar input set $\mathcal{I} = \langle c, V, P\rangle$ is another ImageStar $\mathcal{I}' = \langle c', V', P\rangle$ where $c' = Convol(c)$ is the convolution operation applied to the anchor image, $V' = \{v'_1, \ldots, v'_m\}, v'_i = ConvolZeroBias(v_i)$ is the convolution operation with zero bias applied to the generator images, i.e., only using the weights of the layer.*

Proof. Any image in the ImageStar input set is a *linear* combination of the center and basis images. For any filter in the layer, the convolution operation applied to the input image performs local element-wise multiplication of a local matrix (of all channels) containing the values of the local pixels of the image and the weights of the filter and then combine the result with the bias to get the output for that local region. Due to the linearity of the input image, we can perform the convolution operation with the bias on the center and the convolution operation with zero bias on the basis images and then combine the result to get the output image.

Example 2 (Reachable set of a convolutional layer). The reachable set of a convolutional layer with single 2×2 filter and the ImageStar input set in Example 1 is described in Fig. 2, where the weights and the bias of the filter are $W = \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$, $b = -1$ respectively, the stride is $S = [2\ 2]$, the padding size is $P = [0\ 0\ 0\ 0]$ and the dilation factor is $D = [1\ 1]$.

4.2 Reachability of an Average Pooling Layer

The reachability of an average pooling layer with pooling size PS, padding size P, and stride S is given below, with its proof similar to that of the convolutional layer.

Lemma 2. *The reachable set of a average pooling layer with an ImageStar input set $\mathcal{I} = \langle c, V, P\rangle$ is another ImageStar $\mathcal{I}' = \langle c', V', P\rangle$ where $c' = average(c)$, $V' = \{v'_1, \ldots, v'_m\}, v'_i = average(v_i)$, $average(\cdot)$ is the average pooling operation applied to the anchor and generator images.*

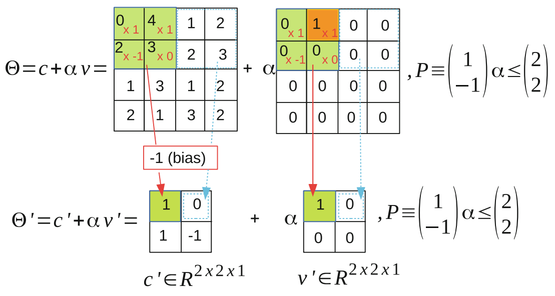


Fig. 2. Reachability of convolutional layer using ImageStar.

Example 3 (Reachable set of an average pooling layer). The reachable set of an 2×2 average pooling layer with padding size $P = [0\ 0\ 0\ 0]$, stride $S = [2\ 2]$, and an ImageStar input set given by Example 1 is shown in Fig. 3.

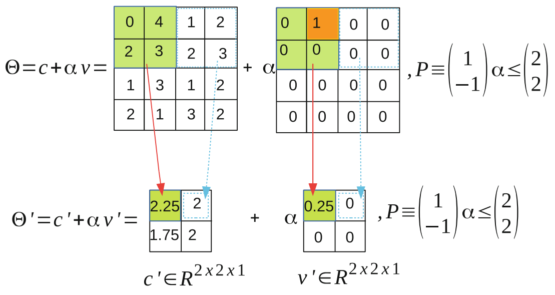


Fig. 3. Reachability of average pooling layer using ImageStar.

4.3 Reachability of a Fully Connected Layer

The reachability of a fully connected layer is stated in the following lemma.

Lemma 3. *Given a two-dimensional fully connected layer with weight* $W_{fc} \in \mathbb{R}^{n_{fc} \times m_{fc}}$*, bias* $b_{fc} \in \mathbb{R}^{n_{fc}}$*, and an ImageStar input set* $\mathcal{I} = \langle c, V, P \rangle$*, the reachable set of the layer is another ImageStar* $\mathcal{I}' = \langle c', V', P \rangle$ *where* $c' = W * \bar{c} + b$*,* $V' = \{v'_1, \ldots, v'_m\}, v'_i = W_{fc} * \bar{v}_i$*,* $\bar{c}(\bar{v}_i) = reshape(c(v_i), [m_{fc}, 1])$*. Note that it is required for consistency between the ImageStar and the weight matrix that* $m_{fc} = h \times w \times nc$*, where* h, w, nc *are the height, width and number of channels of the ImageStar.*

Proof. Similar to the convolutional layer and the average pooling layer, for any image in the ImageStar input set, the fully connected layer performs an affine mapping of the input image which is a linear combination of the center and the basis images of the ImageStar. Due to the linearity, the affine mapping of the input image can be decomposed into the affine mapping of the center image and the affine mapping without the bias of the basis images. The final result is the sum of the individual affine maps.

4.4 Reachability of a Batch Normalization Layer

In the prediction phase, a batch normalization layer normalizes each input channel x_i using the mean μ and variance σ^2 over the full training set. Then the batch normalization layer further shifts and scales the activations using the offset β and the scale factor γ that are learnable parameters. The formula for normalization is as follows:

$$\bar{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y_i = \gamma \bar{x}_i + \beta,$$

where ϵ is a used to prevent division by zero. The batch normalization layer can be described as a tuple $\mathcal{B} = \langle \mu, \sigma^2, \epsilon, \gamma, \beta \rangle$. The reachability of a batch normalization layer with an ImageStar input set is given in the following lemma.

Lemma 4. *The reachable set of a batch normalization layer $\mathcal{B} = \langle \mu, \sigma^2, \epsilon, \gamma, \beta \rangle$ with an ImageStar input set $\mathcal{I} = \langle c, V, P \rangle$ is another ImageStar $\mathcal{I}' = \langle c', V', P' \rangle$ where:*

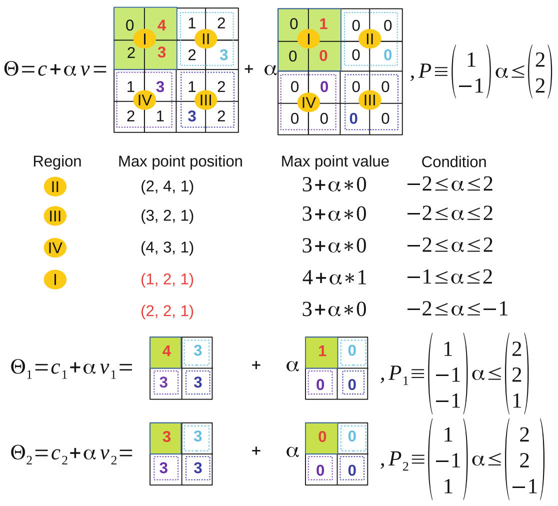
$$c' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} c + \beta - \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \mu, \quad V' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} V, \quad P' \equiv P.$$

The reachable set of a batch normalization layer can be obtained in a straightforward fashion using two affine mappings of the ImageStar input set.

4.5 Reachability of a Max Pooling Layer

Reachability of max pooling layer with an ImageStar input set is challenging because the value of each pixel in an image in the ImageStar depends on the predicate variables α_i. Therefore, the local max point when applying max-pooling operation may change with the values of the predicate variables. In this section, we investigate the exact reachability and over-approximate reachability of a max pooling layer with an ImageStar input set. The first obtains the exact reachable set while the second constructs an over-approximate reachable set.

Exact Reachability of a Max Pooling Layer. The central idea in the exact analysis of the max-pooling layer is finding a set of *local max point candidates* when we apply the max pooling operation on the image. We consider the max pooling operation on the ImageStar in Example 1 with a pool size of 2×2, a padding size of $P = [0\ 0\ 0\ 0]$, and a stride $S = [2\ 2]$ to clarify the exact analysis step-by-step. First, the max-pooling operation is applied on 4 local



Region	Max point position	Max point value	Condition
II	(2, 4, 1)	$3+\alpha*0$	$-2\leq\alpha\leq 2$
III	(3, 2, 1)	$3+\alpha*0$	$-2\leq\alpha\leq 2$
IV	(4, 3, 1)	$3+\alpha*0$	$-2\leq\alpha\leq 2$
I	(1, 2, 1)	$4+\alpha*1$	$-1\leq\alpha\leq 2$
	(2, 2, 1)	$3+\alpha*0$	$-2\leq\alpha\leq -1$

Fig. 4. Exact reachability of max pooling layer using ImageStars.

regions I, II, III, IV, as shown in Fig. 4. The local regions II, III, IV have only one *max point candidate* whic is the pixel that has the maximum value in the region. It is interesting to note that region I has two max point candidates at the positions $(1, 2, 1)$ and $(2, 2, 1)$ and these candidates correspond to different conditions of the predicate variable α. For example, the pixel at the position $(1, 2, 1)$ is the max point if and only if $4 + \alpha \times 1 \geq 3 + \alpha \times 0$. Note that with $-2 \leq \alpha \leq 2$, we always have $4 + \alpha * 1 \geq 2 + \alpha \times 0 \geq 0 + \alpha \times 0$. Since the local region I has two max point candidates, and other regions have only one, the exact reachable set of the max-pooling layer is the union of two new ImageStars Θ_1 and Θ_2. In the first reachable set Θ_1, the max point of the region I is $(1, 2, 1)$ with an additional constraint on the predicate variable $\alpha \geq -1$. For the second reachable set Θ_2, the max point of the region I is $(2, 2, 1)$ with an additional constraint on the predicate variable $\alpha \leq -1$. One can see that from a single ImageStar input set, the output reachable set of the max-pooling layer is split into two new ImageStars. Therefore, the number of ImageStars in the reachable set of the max-pooling layer may grow quickly if each local region has more than one max point candidates. The worst-case complexity of the number of ImageStars in the exact reachable set of the max-pooling layer is given in Lemma 5. The exact reachability algorithm is presented in the Appendix of the extended version of this paper [33].

Lemma 5. *The worst-case complexity of the number of ImageStars in the exact reachability of the max pooling layer is* $\mathcal{O}(((p_1 \times p_2)^{h \times w})^{nc})$ *where* $[h, w, nc]$ *is the size of the ImageStar output sets, and* $[p_1, p_2]$ *is the size of the max-pooling layer.*

Proof. An image in the ImageStar output set has $h \times w$ pixels in each channel. For each pixel, in the worst case, there are $p_1 \times p_2$ candidates. Therefore, the number of ImageStars in the output set in the worst case is $\mathcal{O}(((p_1 \times p_2)^{h \times w})^{nc})$.

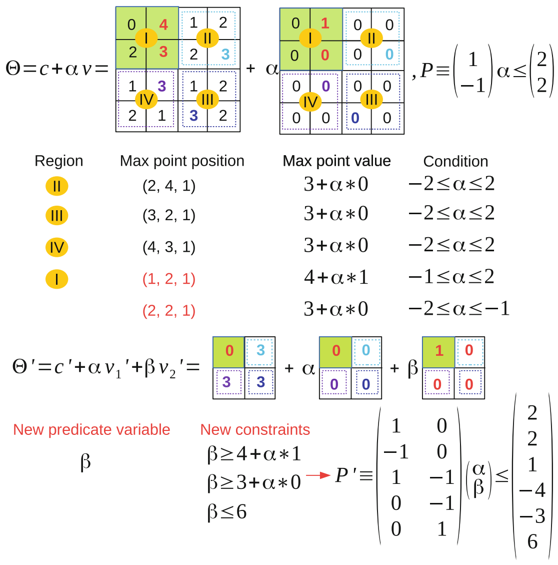


Fig. 5. Over-approximate reachability of max pooling layer using ImageStar.

Finding a set of local max point candidates is the core computation in the exact reachability of max-pooling layer. To optimize this computation, we divide the search for the local max point candidates into two steps. The first one is to estimate the ranges of all pixels in the ImageStar input set. We can solve $h_I \times w_I \times nc$ linear programming optimizations to find the exact ranges of these pixels, where $[h_I, w_I, nc]$ is the size of the input set. However, unfortunately this is a time-consuming computation. For example, ***if a single linear optimization can be done in*** 0.01 s, ***for an ImageStar of the size*** $224 \times 224 \times 32$***, we need about*** 10 **h** ***to find the ranges of all pixels***. To overcome this bottleneck, we quickly estimate the ranges using only the ranges of the predicate variables to get rid of a vast amount of non-max-point candidates. In the second step, we solve a much smaller number of LP optimizations to determine the exact set of the local max point candidates and then construct the ImageStar output set based on these candidates.

Lemma 5 shows that the number of ImageStars in the exact reachability analysis of a max-pooling layer may grow exponentially. To overcome this problem, we propose the following over-approximate reachability method.

Over-Approximate Reachability of a Max Pooling Layer. The central idea of the over-approximate analysis of the max-pooling layer is that if a local region has more than one max point candidates, we introduce a *new predicate variable* standing for the max point of that region. We revisit the example introduced earlier in the exact analysis to clarify this idea. Since the first local region I has two max point candidates, we introduce new predicate variable β to represent the max point of this region by adding three new constraints: 1) $\beta \geq 4 + \alpha * 1$, i.e., β must be equal or larger than the value of the first candidate ; 2) $\beta \geq 3 + \alpha * 0$, i.e., β must be equal or larger than the value of the second candidate; 3) $\beta \leq 6$, i.e., β must be equal or smaller than the upper bound of the pixels values in the region. With the new predicate variable, a single over-approximate reachable set Θ' can be constructed in Fig. 5. The approximate reachability algorithm is presented in the Appendix of the extended version of this paper [33].

Lemma 6. *The worst-case complexity of the new predicate variables introduced in the over-approximate analysis is* $\mathcal{O}(h \times w \times nc)$ *where* $[h, w, nc]$ *is the size of the ImageStar output set.*

4.6 Reachability of a ReLU Layer

Similar to max-pooling layer, the reachability analysis of a ReLU layer is also challenging because the value of each pixel in an ImageStar may be smaller than zero or larger than zero depending on the values of the predicate variables ($ReLU(x) = max(0, x)$). In this section, we investigate the exact and over-approximate reachability algorithms for a ReLU layer with an ImageStar input set. The techniques we use in this section are adapted from in [37].

Exact Reachability of a ReLU Layer. The central idea of the exact analysis of a ReLU layer with an ImageStar input set is performing a sequence of *stepReLU operations* over all pixels of the ImageStar input set. Mathematically, the exact reachable set of a ReLU layer L can be computed as follows.

$$\mathcal{R}_L = stepReLU_N(stepReLU_{N-1}(\dots(stepReLU_1(\mathcal{I})))),$$

where N is the total number of pixels in the ImageStar input set $\mathcal{I}$. The $stepReLU_i$ operation determines whether or not a split occurs at the i^{th} pixel. If the pixel value is larger than zero, then the output value of that pixel remains the same. If the pixel value is smaller than zero than the output value of that pixel is reset to be zero. The challenge is that the pixel value depends on the predicate variables. Therefore, there is the case that the pixel value may be negative or positive with *an extra condition* on the predicate variables. In this case, we split the input set into two *intermediate* ImageStar reachable sets and apply the ReLU law on each intermediate reach set. An example of the stepReLU operation on an ImageStar is illustrated in Fig. 6. The value of the first pixel value $-1 + \alpha$ would be larger than zero if $\alpha \leq 1$, and in this case we have $ReLU(-1 + \alpha) = -1 + \alpha$. If $\alpha <= 1$, then $ReLU(-1 + \alpha) = 0 + \alpha \times 0$. Therefore,

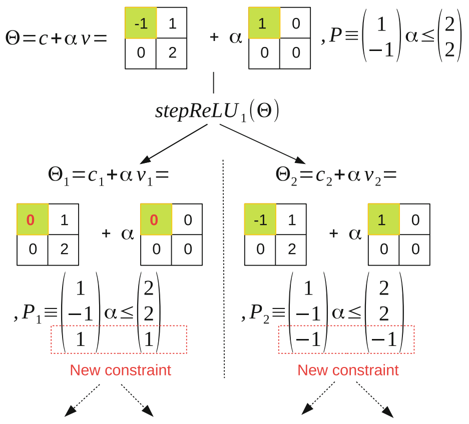


Fig. 6. stepReLU operation on an ImageStar.

the first stepReLU operation produces two intermediate reachable sets Θ_1 and Θ_2, as shown in the figure. The number of ImageStars in the exact reachable set of a ReLU layer increases quickly along with the number of splits in the analysis, as stated in the following lemma.

Lemma 7. *The worst-case complexity of the number of ImageStars in the exact analysis of a ReLU layer is $\mathcal{O}(2^N)$, where N is the number of pixels in the ImageStar input set.*

Proof. There are $h \times w \times nc$ local regions in the approximate analysis. In the worst case, we need to introduce a new variable for each region. Therefore, the worst case complexity of new predicate variables introduced is $\mathcal{O}(h \times w \times nc)$.

Similar to [37], to control the explosion in the number of ImageStars in the exact reachable set of a ReLU layer, we propose an over-approximate reachability algorithm in the following.

Over-Approximate Reachability of a ReLU Layer. The idea behind the over-approximate reachability of ReLU layer is replacing the stepReLU operation at each pixel in the ImageStar input set by an *approxStepReLU* operation. At each pixel where a split occurs, we introduce a new predicate variable to over-approximate the result of the stepReLU operation at that pixel. An example of the overStepReLU operation on an ImageStar is depicted in Fig. 7 in which the first pixel of the input set has the ranges of $[l_1 = -3, u_1 = 1]$ indicating that a split occurs at this pixel. To avoid this split, we introduce a new predicate variable β to over-approximate the exact intermediate reachable set (i.e., two blue segments in the figure) by a triangle. This triangle is determined by three constraints: 1) $\beta \geq 0$ (the $ReLU(x) \geq 0$ for any x); 2) $\beta \geq -1+\alpha$ ($ReLU(x) \geq x$ for any x); 3) $\beta \leq 0.5+0.25\alpha$ (upper bound of the new predicate variable). Using

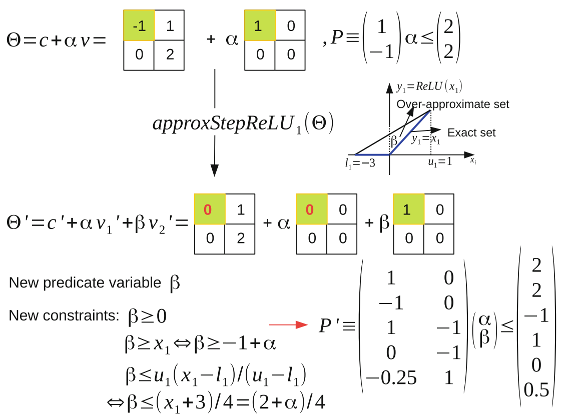


Fig. 7. approxStepReLU operation on an ImageStar.

this over-approximation, a single intermediate reachable set Θ' is produced as shown in the figure. After performing a sequence of approxStepReLU operations, we obtain a single over-approximate ImageStar reachable set for the ReLU layer. However, the number of predicate variables and the number of constraints in the obtained reachable set increase.

Lemma 8. *The worst case complexity of the increment of predicate variables and constraints is $\mathcal{O}(N)$ and $\mathcal{O}(3 \times N)$ respectively, where N is the number of pixels in the ImageStar input set.*

Proof. In the worst case, splits occur at all N pixels in the ImageStar input set. In this case, we need to introduce N new predicate variables to over-approximate the exact intermediate reachable set. For each new predicate variable, we add 3 new constraints.

One can see that determining where splits occur is crucial in the exact and over-approximate analysis of a ReLU layer. To do this, we need to know the ranges of all pixels in the ImageStar input set. However, as mentioned earlier, the computation of the exact range is expensive. To reduce the computation cost, we first use the estimated ranges of all pixels to remove a vast amount of non-splitting pixels. Then, we compute exact ranges for the pixels where splits may occur to compute the exact or over-approximate reachable set of the layer.

4.7 Reachabilty Algorithm and Parallelization

We have presented the core ideas for reachability analysis of different types of layers in a CNN. The reachable set of a CNN is constructed layer-by-layer in which the output reachable set of the previous layer is the input for the next layer. For the convolutional layer, average pooling layer and fully connected layer, we always can compute efficiently the exact reachable set of each layer.

Algorithm 1. Reachability analysis for a CNN.

```
Input: $\mathcal{N} = \{L_i\}_1^n$, $\mathcal{I}$, $scheme$ ('exact' or 'approx')
Output: $R_{\mathcal{N}}$
1: procedure $R_{\mathcal{N}}$ = REACH($\mathcal{N}, \mathcal{I}, scheme$)
2:     $In = \mathcal{I}$
3:     parfor $i = 1 : n$ do $In = L_i.reach(In, scheme)$
4:     end parfor
5:     $R_{\mathcal{N}} = In$
```

For the max pooling layer and ReLU layer, we can compute both the exact and the over-approximate reachable sets. However, the number of ImageStars in the exact reachable set may grow quickly. Therefore, ***in the exact analysis, a layer may receive multiple input sets which can be handled in parallel to speed up the computation time***. The reachability algorithm for a CNN is summarized in Algorithm 1. The detailed implementation of the reachability algorithm for each layer can be found in NNV [34,39].

5 Evaluation

The proposed reachability algorithms are implemented in NNV [39], a tool for verification of deep neural networks and learning-enabled CPS. NNV utilizes core functions in MatConvNet [40] for the analysis of several layers. The evaluation of our approach consists of two parts. First, we evaluate robustness verification of deep neural networks in comparison to zonotope [29] and polytope methods [30] that are re-implemented in NNV. Second, we evaluate the scalability of our approach and the DeepPoly polytope method using real-world image classifiers, VGG16, and VGG19 [28]. The experiments are done on a computer with following configurations: Intel Core i7-6700 CPU @ 3.4GHz $\times$ 8 Processor, 62.8 GiB Memory, Ubuntu 18.04.1 LTS OS.[1] Lastly, we present a comparison with ERAN-DeepZ method on their $ConvMaxPool$ network trained on the CIFAR-10 data set in the Appendix of the extended version of this paper [33].

5.1 Robustness Verification of MNIST Classification Networks

We compare our approach with the zonotope and polytope methods in two aspects including verification time and conservativeness of the results. To do that, we train 3 CNNs in small, medium, and large architectures with $98\%, 99.7\%$, and 99.9% accuracy, respectively, using the MNIST data set consisting of 60000 images of handwritten digits with a resolution of 28×28 pixels [20]. The network architectures are given in the Appendix of the extended version of this paper [33].

[1] Comparison code is available in the NNV repository: https://github.com/verivital/nnv/tree/cav2020imagestar/code/nnv/examples/Submission/CAV2020_ImageStar and as a CodeOcean capsule [34]: https://doi.org/10.24433/CO.3351375.v1.

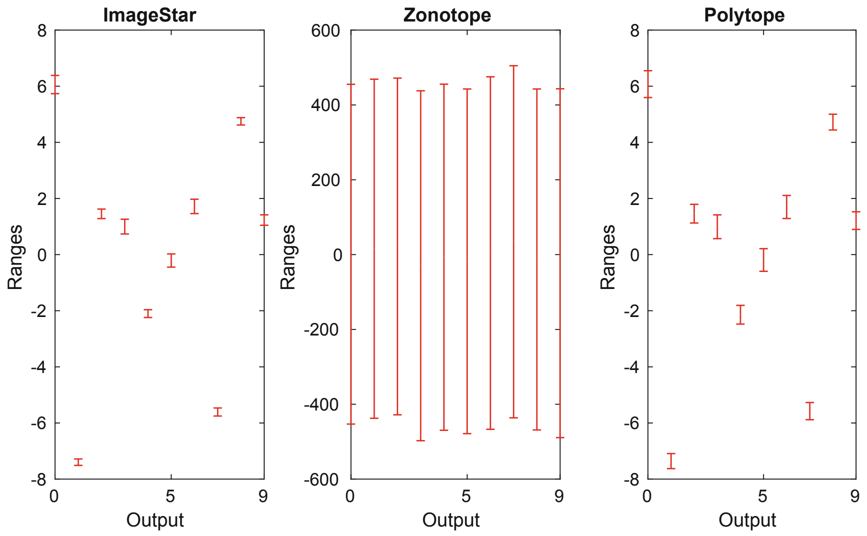


Fig. 8. Example output ranges of the small MNIST classification network using different approaches.

The networks classify images into ten classes: $0, 1, \ldots, 9$. The classified output is the index of the dimension that has maximum value, i.e., the argmax across the 10 outputs. We evaluate the robustness of the network under the well-known brightening attack used in [8]. The idea of a brightening attack is that we can change the value of some pixels independently in the image to make it brighter or darker to fool the network, to misclassify the image. In this case study, we darken a pixel of an image if its value x_i (between 0 and 255) is larger than a threshold d, i.e., $x_i \geq d$. Mathematically, we reduce the value of that pixel x_i to the new value x'_i such that $0 \leq x'_i \leq \delta \times x_i$.

The robustness verification is done as follows. We select 100 images that are correctly classified by the networks and perform the brightening attack on these, which are then used to evaluate the robustness of the networks. A network is robust to an input set if, for any *attacked* image, this is correctly classified by the network. We note that the input set contains an infinite number of images. Therefore, to prove the robustness of the network to the input set, we first compute the output set containing all possible output vectors of the network using reachability analysis. Then, we prove that in the output set, the correctly classified output always has the maximum value compared with other outputs. Note that we can neglect the *softmax* and *classoutput* layers of the networks in the analysis since we only need to know the maximum output in the output set of the last fully connected layer in the networks to prove the robustness of the network.

We are interested in the percentage of the number of input sets that a network is provably robust and the verification times of different approaches under different values of d and θ. When d is small, the number of pixels in the image that are attacked is large and vice versa. For example, the average number of pixels attacked (computed on 100 cases) corresponding to $d = 250$, 245 and 240 are 15, 21 and 25 respectively. The value of δ dictates the size of the input set that can be created by a specific attack. Stated differently it dictates the range in which the value of a pixel can be changed. For example, if $d = 250$ and $\delta = 0.01$, the value of an attacked pixel many range from 0 to 2.55.

Table 1. Verification results of the small MNIST CNN.

	Robustness results (in Percent)					
	$\delta = 0.005$		$\delta = 0.01$		$\delta = 0.015$	
	Polytope	*ImageStar*	*Polytope*	*ImageStar*	*Polytope*	*ImageStar*
$d = 250$	86.00	87.00	84.00	87.00	83.00	87.00
$d = 245$	77.00	78.00	72.00	78.00	70.00	77.00
$d = 240$	72.00	73.00	67.00	72.00	65.00	71.00
	Verification times (in Seconds)					
$d = 250$	11.24	16.28	18.26	28.19	26.42	53.43
$d = 245$	14.84	19.44	24.96	40.76	38.94	85.97
$d = 240$	18.29	25.77	33.59	64.10	54.23	118.58

Table 2. Verification results of the medium MNIST CNN.

	Robustness results (in Percent)					
	$\delta = 0.005$		$\delta = 0.01$		$\delta = 0.015$	
	Polytope	*ImageStar*	*Polytope*	*ImageStar*	*Polytope*	*ImageStar*
$d = 250$	86.00	99.00	73.00	99.00	65.00	99.00
$d = 245$	74.00	95.00	58.00	95.00	46.00	95.00
$d = 240$	69.00	90.00	49.00	89.00	38.00	88.00
	Verification times (in Seconds)					
$d = 250$	213.86	52.09	627.14	257.12	1215.86	749.41
$d = 245$	232.81	68.98	931.28	295.54	2061.98	1168.31
$d = 240$	301.58	102.61	1451.39	705.03	3148.16	2461.89

The experiments show that using the zonotope method, we cannot prove the robustness of any network. The reason is that the zonotope method obtains very conservative reachable sets. Figure 8 illustrates the ranges of the outputs computed by our ImageStar (approximate scheme), the zonotope and polytope approaches when we attack a digit 0 image with brightening attack in which $d = 250$ and $\delta = 0.05$. One can see that, using ImageStar and polytope method, we can prove that the output corresponding to the digit 0 is the one that has a maximum value, which means that the network is robust in this case. However, the zonotope method produces very large output ranges that cannot be used to prove the robustness of the network. The figure also shows that our ImageStar method produces tighter ranges than the polytope method, which means our result is less conservative than the one obtained by the polytope method. We note that the zonotope method is very time-consuming. It needs 93 s to compute the reachable set of the network in this case, while the polytope method only needs 0.3 s, and our approximate ImageStar method needs 0.74 s. The main reason is that the zonotope method introduces many new variables when constructing the reachable set of the network, which results in the increase in both computation time and conservativeness.

Table 3. Verification results of the large MNIST CNN.

	Robustness results (in Percent)					
	$\delta = 0.005$		$\delta = 0.01$		$\delta = 0.015$	
	Polytope	*ImageStar*	*Polytope*	*ImageStar*	*Polytope*	*ImageStar*
$d = 250$	90.00	99.00	83.00	99.00	*MemErr*	99.00
$d = 245$	91.00	100.00	75.00	100.00	*MemErr*	100.00
$d = 240$	81.00	99.00	*MemErr*	99.00	*MemErr*	99.00
	Verification times (in Seconds)					
$d = 250$	917.23	67.45	5221.39	231.67	*MemErr*	488.69
$d = 245$	1420.58	104.71	6491.00	353.02	*MemErr*	1052.87
$d = 240$	1872.16	123.37	*MemErr*	476.67	*MemErr*	1522.50

The comparison of the polytope and our ImageStar method is given in Tables 1, 2, and 3. The tables show that in all networks, our method is less conservative than the polytope approach since the number of cases that our approach can prove the robustness of the network is larger than the one proved by the polytope method. For example, for the small network, for $d = 240$ and $\delta = 0.015$, we can prove 71 cases while the polytope method can prove 65 cases. Importantly, the number of cases proved by DeepPoly reduces quickly when the network becomes larger. For example, for the case that $d = 240$ and $\delta = 0.015$, the polytope method is able to prove the robustness of the medium network for 38 cases while our approach can prove 88 cases. This is because the polytope method becomes more and more conservative when the network or the input set is large. The tables show that the polytope method is faster than our ImageStar method on the small network. However, it is slower than the ImageStar method on any larger networks in all cases. Notably, for the large network, the ImageStar approach is significantly faster than the polytope approach, 16.65 times faster in average. The results also show that the polytope approach may run into memory problem for some large input sets.

Table 4. Verification results of VGG networks.

	Robustness results (in percentage)							
	VGG16				VGG19			
	$\delta = 10^{-7}$		$\delta = 2 \times 10^{-7}$		$\delta = 10^{-7}$		$\delta = 2 \times 10^{-7}$	
	Polytope	*ImageStar*	*Polytope*	*ImageStar*	*Polytope*	*ImageStar*	*Polytope*	*ImageStar*
$l = 0.96$	85.00	85.00	85.00	85.00	100.00	100.00	100.00	100.00
$l = 0.97$	85.00	85.00	85.00	85.00	100.00	100.00	100.00	100.00
$l = 0.98$	85.00	85.00	85.00	85.00	95.00	95.00	95.00	95.00
	Verification times (in Seconds)							
$l = 0.96$	319.04	318.60	327.61	319.93	320.91	314.14	885.07	339.30
$l = 0.97$	324.93	323.41	317.27	324.90	315.84	315.27	319.67	314.58
$l = 0.98$	315.54	315.26	468.59	332.92	320.53	320.44	325.92	317.95

5.2 Robustness Verification of VGG16 and VGG19

In this section, we evaluate the polytope and ImageStar methods on real-world CNNs, the VGG16 and VGG19 classification networks [28]. We use Foolbox [26] to generate the well-known DeepFool adversarial attacks [25] on a set of 20 bell pepper images. From an original image ori_im, Foolbox generates an adversarial image adv_im that can fool the network. The difference between two images is defined by $diff\_im = adv\_im - ori\_im$. We want to verify if we apply $(l + \delta)$ percent of the attack on the original image, whether or not the network classifies the disturbed images correctly. The set of disturbed images can be represented as an ImageStar as follows $disb\_im = ori\_im + (l + \delta) \times diff\_im$, where l is the percentage of the attack at which we want to verify the robustness of the network, and δ is a small perturbation around l, i.e., $0 \leq \delta \leq \delta_{max}$. Intuitively, l describes how close we are to the attack, and the perturbation δ represents the size of the input set.

Table 4 shows the verification results of VGG16 and VGG19 with different levels of the DeepFool attack. The networks are robust if they classify correctly the set of disturbed images $disb\_im$ as bell peppers. To guarantee the robustness of the networks, the output corresponding to the bell pepper label (index 946) needs to be the maximum output compared with others. The table shows that with a small input set, small δ, the polytope and ImageStar can prove robustness of VGG16 and VGG19 in a reasonable amount of time. Notably, the verification times as well as the robustness results of the polytope and ImageStar methods are similar when they deal with small input sets except for two cases where ImageStar is faster than the polytope method. It is interesting to note that according to the verification results for the VGG and MNIST networks, deep networks may be more robust than shall ow networks.

5.3 Exact Analysis vs. Approximate Analysis

We compare our ImageStar approximate scheme with the zonotope and polytope approximation methods, and investigate the performance of the ImageStar

Table 5. Verification results of the VGG16 and VGG19 in which VT is the verification time (in seconds) using the ImageStar exact and approximate schemes.

l	$\delta_{\mathbf{max}}$	VGG16				VGG19			
		Exact		Approximate		Exact		Approximate	
		Robust	VT	Robust	VT	Robust	VT	Robust	VT
50%	10^{-7}	Yes	64.56226	Yes	60.10607	Yes	234.11977	Yes	72.08723
	2×10^{-7}	Yes	63.88826	Yes	59.48936	Yes	1769.69313	Yes	196.93728
80%	10^{-7}	Yes	64.92889	Yes	60.31394	Yes	67.11730	Yes	63.33389
	2×10^{-7}	Yes	64.20910	Yes	59.77254	Yes	174.55983	Yes	200.89500
95%	10^{-7}	Yes	67.64783	Yes	59.89077	Yes	73.13642	Yes	67.56389
	2×10^{-7}	Yes	63.83538	Yes	59.23282	Yes	146.16172	Yes	121.91447
97%	10^{-7}	Yes	64.30362	Yes	59.79876	Yes	77.25398	Yes	64.43168
	2×10^{-7}	Yes	64.06285	Yes	61.23296	Yes	121.70296	Yes	107.17331
98%	10^{-7}	Yes	64.06183	Yes	59.89959	No	67.68139	Unkown	64.47035
	2×10^{-7}	Yes	64.01997	Yes	59.77469	No	205.00939	Unknown	107.42679
98.999%	10^{-7}	Yes	64.24773	Yes	60.22833	No	71.90568	Unknown	68.25916
	2×10^{-7}	Yes	63.67108	Yes	59.69298	No	106.84492	Unknown	101.04668

exact scheme compared to the approximate one. To illustrate the advantages and disadvantages of the exact scheme and approximate scheme, we consider the robustness verification of VGG16 and VGG19 on a single ImageStar input set created by an adversarial attack on a bell pepper image. The verification results are presented in Table 5. The table shows that for a small perturbation δ, the exact and over-approximate analysis can prove the robustness of the VGG16 around some specific levels of attack in approximately one minute. We can intuitively verify the robustness of the VGG networks via visualization of their output ranges. An example of the output ranges of VGG19 for the case of $l = 0.95\%, \delta_{max} = 2 \times 10^{-7}$ is depicted in Fig. 9. One can see from the figure that the output of the index 946 corresponding to the bell pepper label is always the maximum one compared with others, which proves that VGG19 is robust in this case. From the table, it is interesting that VGG19 is not robust if we apply $\geq$ 98% of the attack. Notably, the exact analysis can give us correct answers with a counter-example set in this case. However, the over-approximate analysis cannot prove that VGG19 is not robust since its obtained reachable set is an over-approximation of the exact one. Therefore, it may be the case that the over-approximate reachable set violates the robustness property because of its conservativeness. A counter-example generated by the exact analysis method is depicted in Fig. 10 in which the disturbed image is classified as strawberry instead of bell pepper since the strawberry output is larger than the bell pepper output in this case.

To optimize the verification time, it is important to know the times consumed by each type of layer in the reachability analysis step. Figure 11 described the total reachability times of the convolutional layers, fully connected layers, max pooling layers and ReLU layers in the VGG19 with 50% attack and 10^{-7} perturbation. As shown in the figure, the reachable set computation in the convo-

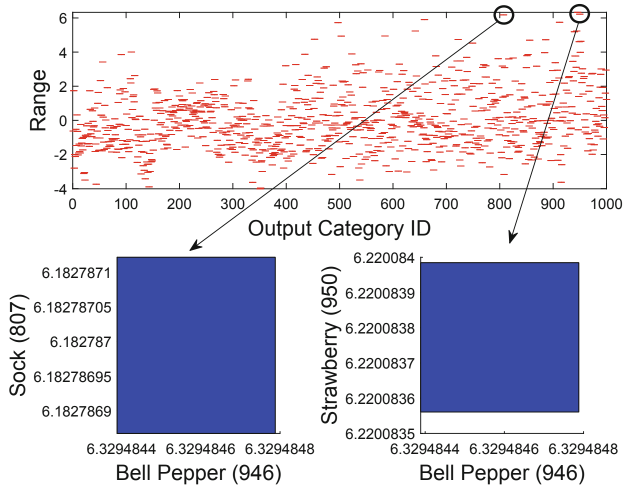


Fig. 9. Exact ranges of VGG19 show that VGG19 correctly classifies the input image as a bell pepper.

lutional layers and fully connected layers can be done very quickly, which shows the advantages of the ImageStar data structure. Notably, the total reachability time is dominated by the time of computing the reachable set for 5 max pooling layers and 18 ReLU layers. This is because the computation in these layers concerns solving a large number of linear programing (LP) optimization problems such as finding lower bound and upper bound, and checking max point candidates. Therefore, to optimize the computation time, we need to minimize the number of LP problems in the future.

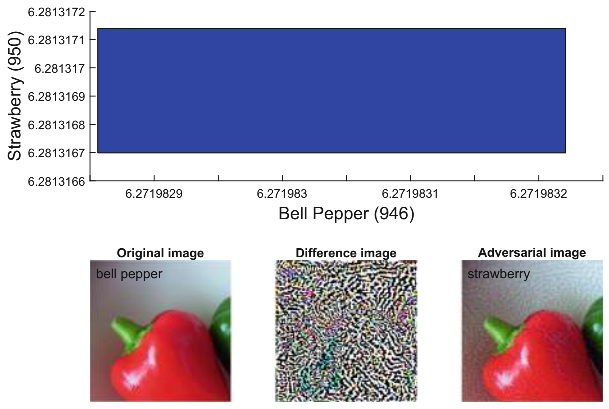


Fig. 10. A counter-example shows that VGG19 misclassifies the input image as a strawberry instead of a bell pepper.

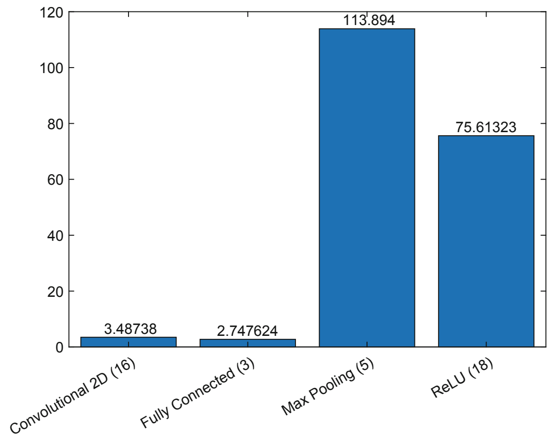


Fig. 11. Total reachability time of each type of layer in VGG19, where the max pooling and ReLU layers dominate the total reachability time.

6 Discussion

When we apply our approach on large networks, it has been shown that the size of the input set is the most important factor that influences the performance of verification approaches. However, this important issue has not been emphasized in the existing literature. Most of existing approaches focus on the size of the network that they can analyze. We hypothesize that existing methods (including the methods in this paper) scalable to large networks are only so for small input sets. When the input set is large, it causes three major problems in the analysis, which are explosions of 1) computation time; 2) memory usage; and 3) conservativeness. In the exact analysis method, a large input set causes more splits in the max-pooling and ReLU layers. A single ImageStar may split into many new ImageStars after these layers, which leads to explosion in the number of ImageStars in the reachable set as shown in Fig. 12. Therefore, it requires more memory to handle the new ImageStars and more time for the computation. One may think that the over-approximate method can overcome this challenge since it obtains only one ImageStar at each layer and at the cost of conservativeness of the result. An over-approximate method does usually help reduce the computation time, as shown in the experimental results. However, it is not necessarily efficient in terms of memory consumption. The reason is, if there is a split, it introduces a new predicate variable and new generator. If the number of generators and the dimensions of the ImageStar are large, it requires a massive amount of memory to store the over-approximate reachable set. For instance, if there are 100 splits in the first ReLU layer of VGG19, the second convolutional layer will receive an ImageStar of size $224 \times 224 \times 64$ with 100 generators. To store this ImageStar with double precision, we need approximately 2.4 GB of memory. In practice, the dimensions of the ImageStars obtained in the first several convolutional layers are usually large. Therefore, if splitting happens in these layers, we may run out of memory. We see that existing approaches, such as those using zonotopes and polytopes, also face the same challenges. Additionally, the conservativeness of an over-approximate reachable set is a crucial factor in evaluating an over-approximation approach. Therefore, the exact analysis still plays an essential role in the analysis of neural networks since it helps to evaluate the conservativeness of the over-approximation approaches.

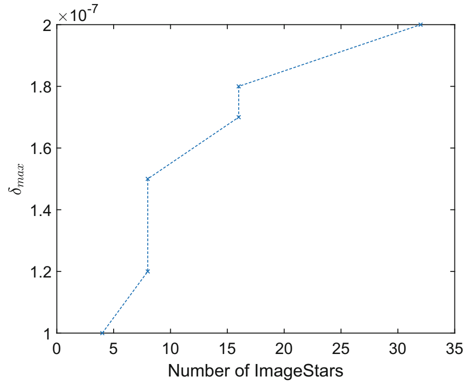


Fig. 12. Number of ImageStars in exact analysis increases with input size.

7 Conclusion

We have presented a new set-based method for robustness verification of deep CNNs using ImageStars. The core of this method are exact and over-approximate reachability algorithms for ImageStar input sets. The experiments show that our approach is less conservative than recent zonotope and polytope approaches. It is also faster than existing approaches when dealing with deep networks. Notably, our approach can be applied to verify the robustness of real-world CNNs with small perturbed input sets. It can also compute the exact reachable set and visualize the exact output range of deep CNNs, and the analysis can speed up significantly with parallel computing. We have found and shown the size of the input set to be an important factor that impacts the performance of reachability algorithms. Future work includes improving the method to deal with larger input sets and optimizing the memory and time complexity of our computations.

References

1. Akintunde, M.E., Botoeva, E., Kouvaros, P., Lomuscio, A.: Formal verification of neural agents in non-deterministic environments. In: Autonomous Agents and Multi-Agent Systems, May 2020

2. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI 2019, pp. 731–744. Association for Computing Machinery, New York (2019)
3. Bak, S., Duggirala, P.S.: Simulation-Equivalent reachability of large linear systems with inputs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 401–420. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_20
4. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 23–32. ACM (2019)
5. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks (2017). arXiv preprint arXiv:1709.09130
6. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI, pp. 550–559 (2018)
7. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
8. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18. IEEE (2018)
9. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples (2014). arXiv preprint arXiv:1412.6572
10. Gopinath, D., Converse, H., Pasareanu, C., Taly, A.: Property inference for deep neural networks. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 797–809, November 2019
11. Hein, M., Andriushchenko, M.: Formal guarantees on the robustness of a classifier against adversarial manipulation. In: Advances in Neural Information Processing Systems, pp. 2266–2276 (2017)
12. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: reachability analysis of neural-network controlled systems. ACM Trans. Embed. Comput. Syst. (TECS) **18**(5s), 1–22 (2019)
13. Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Case study: verifying the safety of an autonomous racing car with a neural network controller. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, pp. 1–7 (2020)
14. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Hybrid Systems: Computation and Control (HSCC) (2019)
15. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
16. Katz, G.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
17. Kouvaros, P., Lomuscio, A.: Formal verification of CNN-based perception systems (2018). arXiv preprint arXiv:1811.11373

18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105 (2012)
19. Lawrence, S., Giles, C.L., Tsoi, A.C., Back, A.D.: Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Netw. **8**(1), 98–113 (1997)
20. LeCun, Y.: The MNIST database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/
21. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
22. Lin, W., et al.: Robustness verification of classification deep neural networks via linear programming. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 11418–11427 (2019)
23. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks (2017). arXiv preprint arXiv:1706.07351
24. Lopez, D.M., Musau, P., Tran, H.D., Johnson, T.T.: Verification of closed-loop systems with neural network controllers. In: Frehse, G., Althoff, M. (eds.) ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 61, pp. 201–210. EasyChair, April 2019
25. Moosavi-Dezfooli, S.M., Fawzi, A., Frossard, P.: Deepfool: a simple and accurate method to fool deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2574–2582 (2016)
26. Rauber, J., Brendel, W., Bethge, M.: Foolbox v0. 8.0: A python toolbox to benchmark the robustness of machine learning models, 5 (2017). arXiv preprint arXiv:1707.04131
27. Ruan, W., Wu, M., Sun, Y., Huang, X., Kroening, D., Kwiatkowska, M.: Global robustness evaluation of deep neural networks with provable guarantees for the l_0 norm (2018). arXiv preprint arXiv:1804.05805
28. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). arXiv preprint arXiv:1409.1556
29. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems, pp. 10825–10836 (2018)
30. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings of the ACM on Programming Languages 3(POPL), 41 (2019)
31. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Hybrid Systems: Computation and Control (HSCC) (2019)
32. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Hybrid Systems: Computation and Control (HSCC) (2019)
33. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars (2020). arXiv preprint arXiv:2004.05511
34. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars (CodeOcean Capsule), May 2020. https://doi.org/10.24433/CO.3351375.v1
35. Tran, H.D., Cei, F., Lopez, D.M., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. In: ACM SIGBED International Conference on Embedded Software (EMSOFT 2019). ACM, October 2019

36. Tran, H.D., et al.: Parallelizable reachability analysis algorithms for feed-forward neural networks. In: 7th International Conference on Formal Methods in Software Engineering (FormaliSE 2019), Montreal, Canada (2019)
37. Tran, H.D., et al.: Star-based reachability analysis of deep neural networks. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 670–686. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_39
38. Tran, H.-D., Nguyen, L.V., Hamilton, N., Xiang, W., Johnson, T.T.: Reachability analysis for high-index linear differential algebraic equations. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 160–177. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29662-9_10
39. Tran, H.D., et al.: NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: 32nd International Conference on Computer-Aided Verification (CAV), July 2020
40. Vedaldi, A., Lenc, K.: Matconvnet: convolutional neural networks for matlab. In: Proceedings of the 23rd ACM international conference on Multimedia, pp. 689–692. ACM (2015)
41. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals (2018). arXiv preprint arXiv:1804.10829
42. Weng, T.W., et al.: Towards fast computation of certified robustness for relu networks (2018). arXiv preprint arXiv:1804.09699
43. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope (2017). arXiv preprint arXiv:1711.00851
44. Wu, M., Wicker, M., Ruan, W., Huang, X., Kwiatkowska, M.: A game-basedapproximate verification of deep neural networks with provable guarantees. Theor. Comput. Sci. (2019)
45. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)
46. Xiang, W., Tran, H.D., Yang, X., Johnson, T.T.: Reachable set estimation for neural network control systems: A simulation-guided approach. IEEE Transactions on Neural Networks and Learning Systems, pp. 1–10 (2020)
47. Xiang, W., Tran, H.D., Johnson, T.T.: Reachable set computation and safety verification for neural networks with relu activations (2017). arXiv preprint arXiv:1712.08163
48. Xiang, W., Tran, H.D., Johnson, T.T.: Specification-guided safety verification for feedforward neural networks. In: AAAI Spring Symposium on Verification of Neural Networks (2019)
49. Xiang, W., Tran, H.D., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. arXiv preprint arXiv:1802.06981 (2018)
50. Yang, X., Tran, H.D., Xiang, W., Johnson, T.T.: Reachability analysis for feed-forward neural networks using face lattices (2020). https://arxiv.org/abs/2003.01226
51. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems, pp. 4944–4953 (2018)

An Abstraction-Based Framework for Neural Network Verification

Yizhak Yisrael Elboher[1], Justin Gottschlich[2], and Guy Katz[1](✉)

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
{yizhak.elboher,g.katz}@mail.huji.ac.il
[2] Intel Labs, Santa Clara, USA
justin.gottschlich@intel.com

Abstract. Deep neural networks are increasingly being used as controllers for safety-critical systems. Because neural networks are opaque, certifying their correctness is a significant challenge. To address this issue, several neural network verification approaches have recently been proposed. However, these approaches afford limited scalability, and applying them to large networks can be challenging. In this paper, we propose a framework that can enhance neural network verification techniques by using over-approximation to reduce the size of the network—thus making it more amenable to verification. We perform the approximation such that if the property holds for the smaller (abstract) network, it holds for the original as well. The over-approximation may be too coarse, in which case the underlying verification tool might return a spurious counterexample. Under such conditions, we perform counterexample-guided refinement to adjust the approximation, and then repeat the process. Our approach is orthogonal to, and can be integrated with, many existing verification techniques. For evaluation purposes, we integrate it with the recently proposed Marabou framework, and observe a significant improvement in Marabou's performance. Our experiments demonstrate the great potential of our approach for verifying larger neural networks.

1 Introduction

Machine programming (MP), the automatic generation of software, is showing early signs of fundamentally transforming the way software is developed [15]. A key ingredient employed by MP is the *deep neural network* (DNN), which has emerged as an effective means to semi-autonomously implement many complex software systems. DNNs are artifacts produced by *machine learning*: a user provides examples of how a system should behave, and a machine learning algorithm generalizes these examples into a DNN capable of correctly handling inputs that it had not seen before. Systems with DNN components have obtained unprecedented results in fields such as image recognition [24], game playing [33], natural language processing [16], computer networks [28], and many others, often surpassing the results obtained by similar systems that have been carefully handcrafted. It seems evident that this trend will increase and intensify, and that DNN components will be deployed in various safety-critical systems [3,19].

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 43–65, 2020.
https://doi.org/10.1007/978-3-030-53288-8_3

DNNs are appealing in that (in some cases) they are easier to create than handcrafted software, while still achieving excellent results. However, their usage also raises a challenge when it comes to certification. Undesired behavior has been observed in many state-of-the-art DNNs. For example, in many cases slight perturbations to correctly handled inputs can cause severe errors [26,35]. Because many practices for improving the reliability of hand-crafted code have yet to be successfully applied to DNNs (e.g., code reviews, coding guidelines, etc.), it remains unclear how to overcome the opacity of DNNs, which may limit our ability to certify them before they are deployed.

To mitigate this, the formal methods community has begun developing techniques for the formal verification of DNNs (e.g., [10,17,20,37]). These techniques can automatically prove that a DNN always satisfies a prescribed property. Unfortunately, the DNN verification problem is computationally difficult (e.g., NP-complete, even for simple specifications and networks [20]), and becomes exponentially more difficult as network sizes increase. Thus, despite recent advances in DNN verification techniques, network sizes remain a severely limiting factor.

In this work, we propose a technique by which the scalability of many existing verification techniques can be significantly increased. The idea is to apply the well-established notion of *abstraction and refinement* [6]: replace a network N that is to be verified with a much smaller, *abstract* network, $\bar{N}$, and then verify this $\bar{N}$. Because $\bar{N}$ is smaller it can be verified more efficiently; and it is constructed in such a way that if it satisfies the specification, the original network N also satisfies it. In the case that $\bar{N}$ does not satisfy the specification, the verification procedure provides a counterexample x. This x may be a true counterexample demonstrating that the original network N violates the specification, or it may be *spurious*. If x is spurious, the network $\bar{N}$ is *refined* to make it more accurate (and slightly larger), and then the process is repeated. A particularly useful variant of this approach is to use the spurious x to guide the refinement process, so that the refinement step rules out x as a counterexample. This variant, known as *counterexample-guided abstraction refinement* (*CEGAR*) [6], has been successfully applied in many verification contexts.

As part of our technique we propose a method for abstracting and refining neural networks. Our basic abstraction step *merges* two neurons into one, thus reducing the overall number of neurons by one. This basic step can be repeated numerous times, significantly reducing the network size. Conversely, refinement is performed by splitting a previously merged neuron in two, increasing the network size but making it more closely resemble the original. A key point is that not all pairs of neurons can be merged, as this could result in a network that is smaller but is not an over-approximation of the original. We resolve this by first transforming the original network into an equivalent network where each node belongs to one of four classes, determined by its edge weights and its effect on the network's output; merging neurons from the same class can then be done safely. The actual choice of which neurons to merge or split is performed heuristically. We propose and discuss several possible heuristics.

For evaluation purposes, we implemented our approach as a Python framework that wraps the Marabou verification tool [22]. We then used our framework to verify properties of the Airborne Collision Avoidance System (ACAS Xu) set of benchmarks [20]. Our results strongly demonstrate the potential usefulness of abstraction in enhancing existing verification schemes: specifically, in most cases the abstraction-enhanced Marabou significantly outperformed the original. Further, in most cases the properties in question could indeed be shown to hold or not hold for the original DNN by verifying a small, abstract version thereof.

To summarize, our contributions are: (i) we propose a general framework for over-approximating and refining DNNs; (ii) we propose several heuristics for abstraction and refinement, to be used within our general framework; and (iii) we provide an implementation of our technique that integrates with the Marabou verification tool and use it for evaluation. Our code is available online [9].

The rest of this paper is organized as follows. In Sect. 2, we provide a brief background on neural networks and their verification. In Sect. 3, we describe our general framework for abstracting an refining DNNs. In Sect. 4, we discuss how to apply these abstraction and refinement steps as part of a CEGAR procedure, followed by an evaluation in Sect. 5. In Sect. 6, we discuss related work, and we conclude in Sect. 7.

2 Background

2.1 Neural Networks

A neural network consists of an *input layer*, an *output layer*, and one or more intermediate layers called *hidden layers*. Each layer is a collection of nodes, called *neurons*. Each neuron is connected to other neurons by one or more directed edges. In a feedforward neural network, the neurons in the first layer receive input data that sets their initial values. The remaining neurons calculate their values using the weighted values of the neurons that they are connected to through edges from the preceding layer (see Fig. 1). The output layer provides the resulting value of the DNN for a given input.

There are many types of DNNs, which may differ in the way their neuron values are computed. Typically, a neuron is evaluated by first computing a weighted sum of the preceding layer's neuron values according to the edge weights, and then applying an activation function to this weighted sum [13]. We focus here on the Rectified Linear Unit (ReLU) activation function [29], given as $\text{ReLU}(x) = \max(0, x)$. Thus, if the weighted sum computation yields a positive value, it is kept; and otherwise, it is replaced by zero.

More formally, given a DNN N, we use n to denote the number of layers of N. We denote the number of nodes of layer i by s_i. Layers 1 and n are the input and output layers, respectively. Layers $2, \ldots, n-1$ are the hidden layers. We denote the value of the j-th node of layer i by $v_{i,j}$, and denote the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$ as V_i.

Evaluating N is performed by calculating V_n for a given input assignment V_1. This is done by sequentially computing V_i for $i = 2, 3, \ldots, n$, each time using

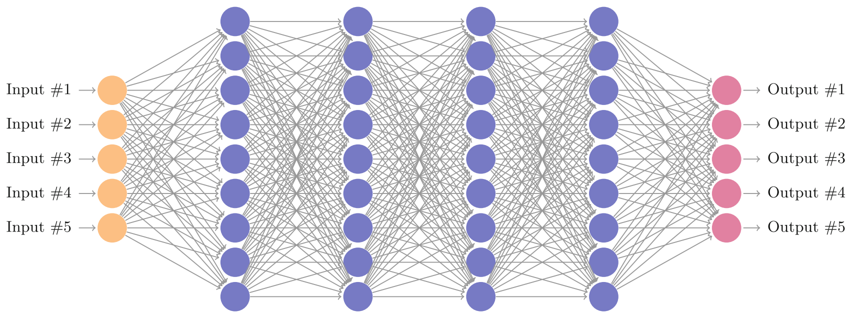


Fig. 1. A fully connected, feedforward DNN with 5 input nodes (in orange), 5 output nodes (in purple), and 4 hidden layers containing a total of 36 hidden nodes (in blue). Each edge is associated with a weight value (not depicted). (Color figure online)

the values of V_{i-1} to compute weighted sums, and then applying the ReLU activation functions. Specifically, layer i (for $i > 1$) is associated with a weight matrix W_i of size $s_i \times s_{i-1}$ and a bias vector B_i of size s_i. If i is a hidden layer, its values are given by $V_i = \text{ReLU}(W_i V_{i-1} + B_i)$, where the ReLUs are applied element-wise; and the output layer is given by $V_n = W_n V_{n-1} + B_n$ (ReLUs are not applied). Without loss of generality, in the rest of the paper we assume that all bias values are 0, and can be ignored. This rule is applied repeatedly once for each layer, until V_n is eventually computed.

We will sometimes use the notation $w(v_{i,j}, v_{i+1,k})$ to refer to the entry of W_{i+1} that represents the weight of the edge between neuron j of layer i and neuron k of layer $i+1$. We will also refer to such an edge as an *outgoing edge* for $v_{i,j}$, and as an *incoming edge* for $v_{i+1,k}$.

As part of our abstraction framework, we will sometimes need to consider a *suffix* of a DNN, in which the first layers of the DNN are omitted. For $1 < i < n$, we use $N^{[i]}$ to denote the DNN comprised of layers $i, i+1, \ldots, n$ of the original network. The sizes and weights of the remaining layers are unchanged, and layer i of N is treated as the input layer of $N^{[i]}$.

Figure 2 depicts a small neural network. The network has $n = 3$ layers, of sizes $s_1 = 1, s_2 = 2$ and $s_3 = 1$. Its weights are $w(v_{1,1}, v_{2,1}) = 1$, $w(v_{1,1}, v_{2,2}) = -1$, $w(v_{2,1}, v_{3,1}) = 1$ and $w(v_{2,2}, v_{3,1}) = 2$. For input $v_{1,1} = 3$, node $v_{2,1}$ evaluates to 3 and node $v_{2,2}$ evaluates to 0, due to the ReLU activation function. The output node $v_{3,1}$ then evaluates to 3.

2.2 Neural Network Verification

DNN verification amounts to answering the following question: given a DNN N, which maps input vector x to output vector y, and predicates P and Q, does there exist an input x_0 such that $P(x_0)$ and $Q(N(x_0))$ both hold? In other words, the verification process determines whether there exists a particular input that meets the input criterion P, and that is mapped to an output that meets the

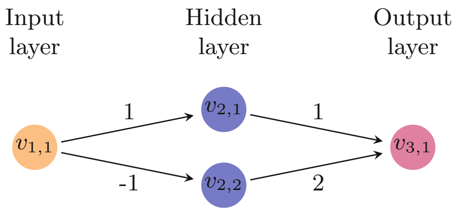


Fig. 2. A simple feedforward neural network.

output criterion Q. We refer to $\langle N, P, Q\rangle$ as the *verification query*. As is usual in verification, Q represents the *negation* of the desired property. Thus, if the query is *unsatisfiable* (`UNSAT`), the property holds; and if it is *satisfiable* (`SAT`), then x_0 constitutes a counterexample to the property in question.

Different verification approaches may differ in (i) the kinds of neural networks they allow (specifically, the kinds of activation functions in use); (ii) the kinds of input properties; and (iii) the kinds of output properties. For simplicity, we focus on networks that employ the ReLU activation function. In addition, our input properties will be conjunctions of linear constraints on the input values. Finally, we will assume that our networks have a single output node y, and that the output property is $y > c$ for a given constant c. We stress that these restrictions are for the sake of simplicity. Many properties of interest, including those with arbitrary Boolean structure and involving multiple neurons, can be reduced into the above single-output setting by adding a few neurons that encode the Boolean structure [20,32]; see Fig. 3 for an example. The number of neurons to be added is typically negligible when compared to the size of the DNN. In particular, this is true for the ACAS Xu family of benchmarks [20], and also for adversarial robustness queries that use the L_∞ or the L_1 norm as a distance metric [5,14,21]. Additionally, other piecewise-linear activation functions, such as max-pooling layers, can also be encoded using ReLUs [5].

Several techniques have been proposed for solving the aforementioned verification problem in recent years (Sect. 6 includes a brief overview). Our abstraction technique is designed to be compatible with most of these techniques, by simplifying the network being verified, as we describe next.

3 Network Abstraction and Refinement

Because the complexity of verifying a neural network is strongly connected to its size [20], our goal is to transform a verification query $\varphi_1 = \langle N, P, Q\rangle$ into query $\varphi_2 = \langle \bar{N}, P, Q\rangle$, such that the abstract network $\bar{N}$ is significantly smaller than N (notice that properties P and Q remain unchanged). We will construct $\bar{N}$ so that it is an over-approximation of N, meaning that if φ_2 is `UNSAT` then φ_1 is also `UNSAT`. More specifically, since our DNNs have a single output, we can regard $N(x)$ and $\bar{N}(x)$ as real values for every input x. To guarantee that φ_2 over-approximates φ_1, we will make sure that for every x, $N(x) \leq \bar{N}(x)$; and

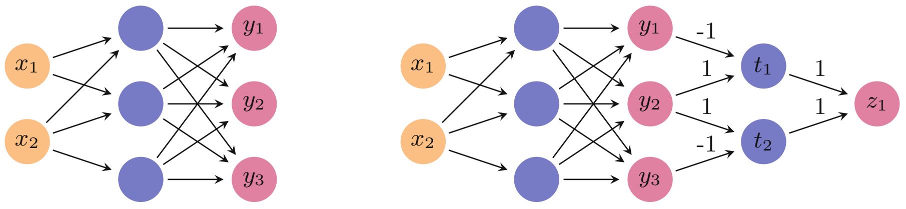


Fig. 3. Reducing a complex property to the $y > 0$ form. For the network on the left hand side, suppose we wish to examine the property $y_2 > y_1 \vee y_2 > y_3$, which is a property that involves multiple outputs and includes a disjunction. We do this (right hand side network) by adding two neurons, t_1 and t_2, such that $t_1 = \text{ReLU}(y_2 - y_1)$ and $t_2 = \text{ReLU}(y_2 - y_3)$. Thus, $t_1 > 0$ if and only if the first disjunct, $y_2 > y_1$, holds; and $t_2 > 0$ if and only if the second disjunct, $y_2 > y_3$, holds. Finally, we add a neuron z_1 such that $z_1 = t_1 + t_2$. It holds that $z_1 > 0$ if and only if $t_1 > 0 \vee t_2 > 0$. Thus, we have reduced the complex property into an equivalent property in the desired form.

thus, $\bar{N}(x) \leq c \implies N(x) \leq c$. Because our output properties always have the form $N(x) > c$, it is indeed the case that if φ_2 is `UNSAT`, i.e. $\bar{N}(x) \leq c$ for all x, then $N(x) \leq c$ for all x and so φ_1 is also `UNSAT`. We now propose a framework for generating various $\bar{N}$s with this property.

3.1 Abstraction

We seek to define an abstraction operator that removes a single neuron from the network, by merging it with another neuron. To do this, we will first transform N into an equivalent network, whose neurons have properties that will facilitate their merging. Equivalent here means that for every input vector, both networks produce the exact same output. First, each hidden neuron $v_{i,j}$ of our transformed network will be classified as either a `pos` neuron or a `neg` neuron. A neuron is `pos` if all the weights on its outgoing edges are positive, and is `neg` if all those weights are negative. Second, orthogonally to the `pos`/`neg` classification, each hidden neuron will also be classified as either an `inc` neuron or a `dec` neuron. $v_{i,j}$ is an `inc` neuron of N if, when we look at $N^{[i]}$ (where $v_{i,j}$ is an input neuron), increasing the value of $v_{i,j}$ increases the value of the network's output. Formally, $v_{i,j}$ is `inc` if for every two input vectors x_1 and x_2 where $x_1[k] = x_2[k]$ for $k \neq j$ and $x_1[j] > x_2[j]$, it holds that $N^{[i]}(x_1) > N^{[i]}(x_2)$. A `dec` neuron is defined symmetrically, so that *decreasing* the value of $x[j]$ *increases* the output. We first describe this transformation (an illustration of which appears in Fig. 4), and later we explain how it fits into our abstraction framework.

Our first step is to transform N into a new network, N', in which every hidden neuron is classified as `pos` or `neg`. This transformation is done by replacing each hidden neuron v_{i_j} with two neurons, $v^+_{i,j}$ and $v^-_{i,j}$, which are respectively `pos` and `neg`. Both $v^+_{i,j}$ an $v^-_{i,j}$ retain a copy of all incoming edges of the original $v_{i,j}$; however, $v^+_{i,j}$ retains just the outgoing edges with positive weights, and $v^-_{i,j}$ retains just those with negative weights. Outgoing edges with negative weights are removed from $v^+_{i,j}$ by setting their weights to 0, and the same is done for outgoing edges with positive weights for $v^-_{i,j}$. Formally, for every neuron $v_{i-1,p}$,

$$w'(v_{i-1,p}, v^+_{i,j}) = w(v_{i-1,p}, v_{i,j}), \qquad w'(v_{i-1,p}, v^-_{i,j}) = w(v_{i-1,p}, v_{i,j})$$

where w' represents the weights in the new network N'. Also, for every neuron $v_{i+1,q}$

$$w'(v^+_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$w'(v^-_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

(see Fig. 4). This operation is performed once for every hidden neuron of N, resulting in a network N' that is roughly double the size of N. Observe that N' is indeed equivalent to N, i.e. their outputs are always identical.

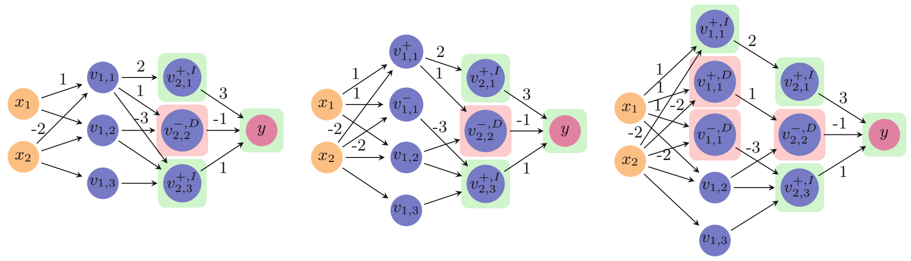


Fig. 4. Classifying neurons as `pos`/`neg` and `inc`/`dec`. In the initial network (left), the neurons of the second hidden layer are already classified: $^+$ and $^-$ superscripts indicate `pos` and `neg` neurons, respectively; the ^{I} superscript and green background indicate `inc`, and the ^{D} superscript and red background indicate `dec`. Classifying node $v_{1,1}$ is done by first splitting it into two nodes $v^+_{1,1}$ and $v^-_{1,1}$ (middle). Both nodes have identical incoming edges, but the outgoing edges of $v_{1,1}$ are partitioned between them, according to the sign of each edge's weight. In the last network (right), $v^+_{1,1}$ is split once more, into an `inc` node with outgoing edges only to other `inc` nodes, and a `dec` node with outgoing edges only to other `dec` nodes. Node $v_{1,1}$ is thus transformed into three nodes, each of which can finally be classified as `inc` or `dec`. Notice that in the worst case, each node is split into four nodes, although for $v_{1,1}$ three nodes were enough.

Our second step is to alter N' further, into a new network N'', where every hidden neuron is either `inc` or `dec` (in addition to already being `pos` or `neg`). Generating N'' from N' is performed by traversing the layers of N' backwards, each time handling a single layer and possibly doubling its number of neurons:

– Initial step: the output layer has a single neuron, y. This neuron is an `inc` node, because increasing its value will increase the network's output value.

– Iterative step: observe layer i, and suppose the nodes of layer $i+1$ have already been partitioned into inc and dec nodes. Observe a neuron $v^{+}_{i,j}$ in layer i which is marked pos (the case for neg is symmetrical). We replace $v^{+}_{i,j}$ with two neurons $v^{+,I}_{i,j}$ and $v^{+,D}_{i,j}$, which are inc and dec, respectively. Both new neurons retain a copy of all incoming edges of $v^{+}_{i,j}$; however, $v^{+,I}_{i,j}$ retains only outgoing edges that lead to inc nodes, and $v^{+,D}_{i,j}$ retains only outgoing edges that lead to dec nodes. Thus, for every $v_{i-1,p}$ and $v_{i+1,q}$,

$$w''(v_{i-1,p}, v^{+,I}_{i,j}) = w'(v_{i-1,p}, v^{+}_{i,j}), \qquad w''(v_{i-1,p}, v^{+,D}_{i,j}) = w'(v_{i-1,p}, v^{+}_{i,j})$$

$$w''(v^{+,I}_{i,j}, v_{i+1,q}) = \begin{cases} w'(v^{+}_{i,j}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{inc} \\ 0 & \text{otherwise} \end{cases}$$

$$w''(v^{+,D}_{i,j}, v_{i+1,q}) = \begin{cases} w'(v^{+}_{i,j}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{dec} \\ 0 & \text{otherwise} \end{cases}$$

where w'' represents the weights in the new network N''. We perform this step for each neuron in layer i, resulting in neurons that are each classified as either inc or dec.

To understand the intuition behind this classification, recall that by our assumption all hidden nodes use the ReLU activation function, which is monotonically increasing. Because $v^{+}_{i,j}$ is pos, all its outgoing edges have positive weights, and so if its assignment was to increase (decrease), the assignments of all nodes to which it is connected in the following layer would also increase (decrease). Thus, we split $v^{+}_{i,j}$ in two, and make sure one copy, $v^{+,I}_{i,j}$, is only connected to nodes that need to increase (inc nodes), and that the other copy, $v^{+,D}_{i,j}$, is only connected to nodes that need to decrease (dec nodes). This ensures that $v^{+,I}_{i,j}$ is itself inc, and that $v^{+,D}_{i,j}$ is dec. Also, both $v^{+,I}_{i,j}$ and $v^{+,D}_{i,j}$ remain pos nodes, because their outgoing edges all have positive weights.

When this procedure terminates, N'' is equivalent to N', and so also to N; and N'' is roughly double the size of N', and roughly four times the size of N. Both transformation steps are only performed for hidden neurons, whereas the input and output neurons remain unchanged. This is summarized by the following lemma:

Lemma 1. *Any DNN N can be transformed into an equivalent network N'' where each hidden neuron is pos or neg, and also inc or dec, by increasing its number of neurons by a factor of at most 4.*

Using Lemma 1, we can assume without loss of generality that the DNN nodes in our input query φ_1 are each marked as pos/neg and as inc/dec. We are now ready to construct the over-approximation network $\bar{N}$. We do this by specifying an abstract operator that merges a pair of neurons in the network (thus reducing network size by one), and can be applied multiple times. The only restrictions are that the two neurons being merged need to be from the same hidden layer, and must share the same `pos`/`neg` and `inc`/`dec` attributes. Consequently, applying `abstract` to saturation will result in a network with at most 4 neurons in each hidden layer, which over-approximates the original network. This, of course, would be an immense reduction in the number of neurons for most reasonable input networks.

The `abstract` operator's behavior depends on the attributes of the neurons being merged. For simplicity, we will focus on the $\langle \texttt{pos}, \texttt{inc} \rangle$ case. Let $v_{i,j}$, $v_{i,k}$ be two hidden neurons of layer i, both classified as $\langle \texttt{pos}, \texttt{inc} \rangle$. Because layer i is hidden, we know that layers $i+1$ and $i-1$ are defined. Let $v_{i-1,p}$ and $v_{i+1,q}$ denote arbitrary neurons in the preceding and succeeding layer, respectively. We construct a network $\bar{N}$ that is identical to N, except that: (i) nodes $v_{i,j}$ and $v_{i,k}$ are removed and replaced with a new single node, $v_{i,t}$; and (ii) all edges that touched nodes $v_{i,j}$ or $v_{i,k}$ are removed, and other edges are untouched. Finally, we add new incoming and outgoing edges for the new node $v_{i,t}$ as follows:

- Incoming edges: $\bar{w}(v_{i-1,p}, v_{i,t}) = \max\{w(v_{i-1,p}, v_{i,j}), w(v_{i-1,p}, v_{i,k})\}$
- Outgoing edges: $\bar{w}(v_{i,t}, v_{i+1,q}) = w(v_{i,j}, v_{i+1,q}) + w(v_{i,k}, v_{i+1,q})$

where $\bar{w}$ represents the weights in the new network $\bar{N}$. An illustrative example appears in Fig. 5. Intuitively, this definition of `abstract` seeks to ensure that the new node $v_{i,t}$ always contributes more to the network's output than the two original nodes $v_{i,j}$ and $v_{i,k}$—so that the new network produces a larger output than the original for every input. By the way we defined the incoming edges of the new neuron $v_{i,t}$, we are guaranteed that for every input x passed into both N and $\bar{N}$, the value assigned to $v_{i,t}$ in $\bar{N}$ is greater than the values assigned to both $v_{i,j}$ and $v_{i,k}$ in the original network. This works to our advantage, because $v_{i,j}$ and $v_{i,k}$ were both `inc`—so increasing their values increases the output value. By our definition of the outgoing edges, the values of any `inc` nodes in layer $i+1$ increase in $\bar{N}$ compared to N, and those of any `dec` nodes decrease. By definition, this means that the network's overall output increases.

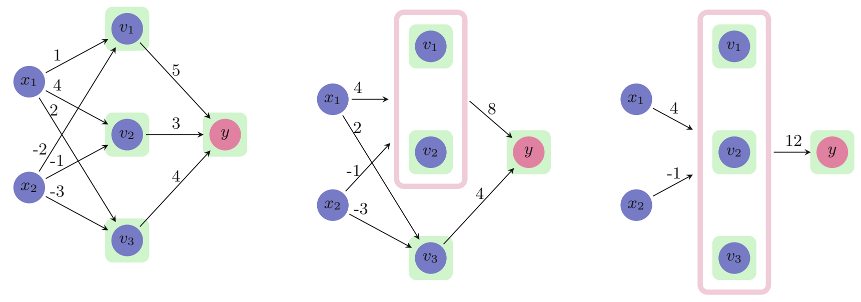
The abstraction operation for the $\langle \texttt{neg}, \texttt{inc} \rangle$ case is identical to the one described above. For the remaining two cases, i.e. $\langle \texttt{pos}, \texttt{dec} \rangle$ and $\langle \texttt{neg}, \texttt{dec} \rangle$, the max operator in the definition is replaced with a min operator.

The next lemma (proof omitted due to lack of space) justifies the use of our abstraction step, and can be applied once per each application of `abstract`:

Lemma 2. *Let $\bar{N}$ be derived from N by a single application of* `abstract`. *For every x, it holds that $\bar{N}(x) \geq N(x)$.*

3.2 Refinement

The aforementioned `abstract` operator reduces network size by merging neurons, but at the cost of accuracy: whereas for some input x_0 the original network returns $N(x_0) = 3$, the over-approximation network $\bar{N}$ created by `abstract` might return $\bar{N}(x_0) = 5$. If our goal is prove that it is never the case that $N(x) > 10$, this over-approximation may be adequate: we can prove that always



$y = 5R(x_1 - 2x_2) + 3R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$ $y = 8R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$ $y = 12R(4x_1 - x_2)$

Fig. 5. Using `abstract` to merge $\langle$`pos`, `inc`$\rangle$ nodes. Initially (left), the three nodes v_1, v_2 and v_3 are separate. Next (middle), `abstract` merges v_1 and v_2 into a single node. For the edge between x_1 and the new abstract node we pick the weight 4, which is the maximal weight among edges from x_1 to v_1 and v_2. Likewise, the edge between x_2 and the abstract node has weight -1. The outgoing edge from the abstract node to y has weight 8, which is the sum of the weights of edges from v_1 and v_2 to y. Next, `abstract` is applied again to merge v_3 with the abstract node, and the weights are adjusted accordingly (right). With every abstraction, the value of y (given as a formula at the bottom of each DNN, where R represents the ReLU operator) increases. For example, to see that $12R(4x_1 - x_2) \geq 8R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$, it is enough to see that $4R(4x_1 - x_2) \geq 4R(2x_1 - 3x_2)$, which holds because ReLU is a monotonically increasing function and x_1 and x_2 are non-negative (being, themselves, the output of ReLU nodes).

$\bar{N}(x) \leq 10$, and this will be enough. However, if our goal is to prove that it is never the case that $N(x) > 4$, the over-approximation is inadequate: it is possible that the property holds for N, but because $\bar{N}(x_0) = 5 > 4$, our verification procedure will return x_0 as a *spurious counterexample* (a counterexample for $\bar{N}$ that is not a counterexample for N). In order to handle this situation, we define a *refinement operator*, `refine`, that is the inverse of `abstract`: it transforms $\bar{N}$ into yet another over-approximation, $\bar{N}'$, with the property that for every x, $N(x) \leq \bar{N}'(x) \leq \bar{N}(x)$. If $\bar{N}'(x_0) = 3.5$, it might be a suitable over-approximation for showing that never $N(x) > 4$. In this section we define the `refine` operator, and in Sect. 4 we explain how to use `abstract` and `refine` as part of a CEGAR-based verification scheme.

Recall that `abstract` merges together a couple of neurons that share the same attributes. After a series of applications of `abstract`, each hidden layer i of the resulting network can be regarded as a partitioning of hidden layer i of the original network, where each partition contains original, *concrete* neurons that share the same attributes. In the abstract network, each partition is represented by a single, *abstract* neuron. The weights on the incoming and outgoing edges of this abstract neuron are determined according to the definition of the `abstract` operator. For example, in the case of an abstract neuron $\bar{v}$ that represents a set of concrete neurons $\{v_1, \ldots, v_n\}$ all with attributes $\langle \texttt{pos}, \texttt{inc} \rangle$, the weight of each incoming edge to $\bar{v}$ is given by

$$\bar{w}(u, v) = \max(w(u, v_1), \ldots, w(u, v_n))$$

where u represents a neuron that has not been abstracted yet, and w is the weight function of the original network. The key point here is that the order of `abstract` operations that merged $v_1, \ldots, v_n$ does not matter—but rather, only the fact that they are now grouped together determines the abstract network's weights. The following corollary, which is a direct result of Lemma 2, establishes this connection between sequences of `abstract` applications and partitions:

Corollary 1. *Let N be a DNN where each hidden neuron is labeled as `pos/neg` and `inc/dec`, and let $\mathcal{P}$ be a partitioning of the hidden neurons of N, that only groups together hidden neurons from the same layer that share the same labels. Then N and $\mathcal{P}$ give rise to an abstract neural network $\bar{N}$, which is obtained by performing a series of `abstract` operations that group together neurons according to the partitions of $\mathcal{P}$. This $\bar{N}$ is an over-approximation of N.*

We now define a `refine` operation that is, in a sense, the inverse of `abstract`. `refine` takes as input a DNN $\bar{N}$ that was generated from N via a sequence of `abstract` operations, and splits a neuron from $\bar{N}$ in two. Formally, the operator receives the original network N, the partitioning $\mathcal{P}$, and a finer partition $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by splitting a single class in two. The operator then returns a new abstract network, $\bar{N}'$, that is the abstraction of N according to $\mathcal{P}'$.

Due to Corollary 1, and because $\bar{N}$ returned by `refine` corresponds to a partition $\mathcal{P}'$ of the hidden neurons of N, it is straightforward to show that $\bar{N}$ is indeed an over-approximation of N. The other useful property that we require is the following:

Lemma 3. *Let $\bar{N}$ be an abstraction of N, and let $\bar{N}'$ be a network obtained from $\bar{N}$ by applying a single `refine` step. Then for every input x it holds that $\bar{N}(x) \geq \bar{N}'(x) \geq N(x)$.*

The second part of the inequality, $\bar{N}'(x) \geq N(x)$ holds because $\bar{N}'$ is an over-approximation of N (Corollary 1). The first part of the inequality, $\bar{N}(x) \geq \bar{N}'(x)$, follows from the fact that $\bar{N}(x)$ can be obtained from $\bar{N}'(x)$ by a single application of `abstract`.

In practice, in order to support the refinement of an abstract DNN, we maintain the current partitioning, i.e. the mapping from concrete neurons to the abstract neurons that represent them. Then, when an abstract neuron is selected for refinement (according to some heuristic, such as the one we propose in Sect. 4), we adjust the mapping and use it to compute the weights of the edges that touch the affected neuron.

4 A CEGAR-Based Approach

In Sect. 3 we defined the `abstract` operator that reduces network size at the cost of reducing network accuracy, and its inverse `refine` operator that increases network size and restores accuracy. Together with a black-box verification procedure *Verify* that can dispatch queries of the form $\varphi = \langle N, P, Q\rangle$, these components now allow us to design an abstraction-refinement algorithm for DNN verification, given as Algorithm 1 (we assume that all hidden neurons in the input network have already been marked `pos`/`neg` and `inc`/`dec`).

Algorithm 1. Abstraction-based DNN Verification(N, P, Q)

```
 1: Use abstract to generate an initial over-approximation $\bar{N}$ of $N$
 2: if $Verify(\bar{N}, P, Q)$ is UNSAT then
 3:     return UNSAT
 4: else
 5:     Extract counterexample $c$
 6:     if $c$ is a counterexample for $N$ then
 7:         return SAT
 8:     else
 9:         Use refine to refine $\bar{N}$ into $\bar{N}'$
10:         $\bar{N} \leftarrow \bar{N}'$
11:         Goto step 2
12:     end if
13: end if
```

Because $\bar{N}$ is obtained via applications of `abstract` and `refine`, the soundness of the underlying *Verify* procedure, together with Lemmas 2 and 3, guarantees the soundness of Algorithm 1. Further, the algorithm always terminates: this is the case because all the `abstract` steps are performed first, followed by a sequence of `refine` steps. Because no additional `abstract` operations are performed beyond Step 1, after finitely many `refine` steps $\bar{N}$ will become identical to N, at which point no spurious counterexample will be found, and the algorithm will terminate with either `SAT` or `UNSAT`. Of course, termination is only guaranteed when the underlying *Verify* procedure is guaranteed to terminate.

There are two steps in the algorithm that we intentionally left ambiguous: Step 1, where the initial over-approximation is computed, and Step 9, where the current abstraction is refined due to the discovery of a spurious counterexample. The motivation was to make Algorithm 1 general, and allow it to be customized by plugging in different heuristics for performing Steps 1 and 9, which may depend on the problem at hand. Below we propose a few such heuristics.

4.1 Generating an Initial Abstraction

The most naïve way to generate the initial abstraction is to apply the `abstract` operator to saturation. As previously discussed, `abstract` can merge together any pair of hidden neurons from a given layer that share the same attributes. Since there are four possible attribute combinations, this will result in each hidden layer of the network having four neurons or fewer. This method, which we refer to as *abstraction to saturation*, produces the smallest abstract networks possible. The downside is that, in some case, these networks might be too coarse, and might require multiple rounds of refinement before a `SAT` or `UNSAT` answer can be reached.

A different heuristic for producing abstractions that may require fewer refinement steps is as follows. First, we select a finite set of input points, $X = \{x_1, \ldots, x_n\}$, all of which satisfy the input property P. These points can be generated randomly, or according to some coverage criterion of the input space. The points of X are then used as indicators in estimating when the abstraction has become too coarse: after every abstraction step, we check whether the property still holds for $x_1, \ldots, x_n$, and stop abstracting if this is not the case. The exact technique, which we refer to as *indicator-guided abstraction*, appears in Algorithm 2, which is used to perform Step 1 of Algorithm 1.

Algorithm 2. Indicator-Guided Abstraction(N, P, Q, X)

```
 1: $\bar{N} \leftarrow N$
 2: while $\forall x \in X.\ \bar{N}(x)$ satisfies $Q$ and there are still neurons that can be merged do
 3:    $\Delta \leftarrow \infty$, bestPair $\leftarrow \bot$
 4:    for every pair of hidden neurons $v_{i,j}, v_{i,k}$ with identical attributes do
 5:       m $\leftarrow 0$
 6:       for every node $v_{i-1,p}$ do
 7:          a $\leftarrow \bar{w}(v_{i-1,p}, v_{i,j})$, b $\leftarrow \bar{w}(v_{i-1,p}, v_{i,k})$
 8:          if $|a - b| >$ m then
 9:             m $\leftarrow |a - b|$
10:          end if
11:       end for
12:       if m $< \Delta$ then
13:          $\Delta \leftarrow$ m, bestPair $\leftarrow \langle v_{i,j}, v_{i,k} \rangle$
14:       end if
15:    end for
16:    Use abstract to merge the nodes of bestPair, store the result in $\bar{N}$
17: end while
18: return $\bar{N}$
```

Another point that is addressed by Algorithm 2, besides how many rounds of abstraction should be performed, is which pair of neurons should be merged in every application of `abstract`. This, too, is determined heuristically. Since any pair of neurons that we pick will result in the same reduction in network size, our strategy is to prefer neurons that will result in a more accurate approximation. Inaccuracies are caused by the $\max$ and $\min$ operators within the `abstract` operator: e.g., in the case of $\max$, every pair of incoming edges with weights a, b are replaced by a single edge with weight $\max(a, b)$. Our strategy here is to merge the pair of neurons for which the *maximal* value of $|a-b|$ (over all incoming edges with weights a and b) is *minimal*. Intuitively, this leads to $\max(a,b)$ being close to both a and b—which, in turn, leads to an over-approximation network that is smaller than the original, but is close to it weight-wise. We point out that although repeatedly exploring all pairs (line 4) may appear costly, in our experiments the time cost of this step was negligible compared to that of the verification queries that followed. Still, if this step happens to become a bottleneck, it is possible to adjust the algorithm to heuristically sample just some of the pairs, and pick the best pair among those considered—without harming the algorithm's soundness.

As a small example, consider the network depicted on the left hand side of Fig. 5. This network has three pairs of neurons that can be merged using `abstract` (any subset of $\{v_1, v_2, v_3\}$). Consider the pair v_1, v_2: the maximal value of $|a-b|$ for these neurons is $\max(|1-4)|, |(-2)-(-1)|) = 3$. For pair v_1, v_3, the maximal value is 1; and for pair v_2, v_3 the maximal value is 2. According to the strategy described in Algorithm 2, we would first choose to apply `abstract` on the pair with the minimal maximal value, i.e. on the pair v_1, v_3.

4.2 Performing the Refinement Step

A refinement step is performed when a spurious counterexample x has been found, indicating that the abstract network is too coarse. In other words, our abstraction steps, and specifically the max and min operators that were used to select edge weights for the abstract neurons, have resulted in the abstract network's output being too great for input x, and we now need to reduce it. Thus, our refinement strategies are aimed at applying `refine` in a way that will result in a significant reduction to the abstract network's output. We note that there may be multiple options for applying `refine`, on different nodes, such that any of them would remove the spurious counterexample x from the abstract network. In addition, it is not guaranteed that it is possible to remove x with a single application of `refine`, and multiple consecutive applications may be required.

One heuristic approach for refinement follows the well-studied notion of counterexample-guided abstraction refinement [6]. Specifically, we leverage the spurious counterexample x in order to identify a concrete neuron v, which is currently mapped into an abstract neuron $\bar{v}$, such that splitting v away from $\bar{v}$ might rule out counterexample x. To do this, we evaluate the original network on x and compute the value of v (we denote this value by $v(x)$), and then do the same for $\bar{v}$ in the abstract network (value denoted $\bar{v}(x)$). Intuitively, a neuron pair $\langle v, \bar{v}\rangle$ for which the difference $|v(x) - \bar{v}(x)|$ is significant makes a good candidate for a refinement operation that will split v away from $\bar{v}$.

In addition to considering $v(x)$ and $\bar{v}(x)$, we propose to also consider the weights of the incoming edges of v and $\bar{v}$. When these weights differ significantly, this could indicate that $\bar{v}$ is too coarse an approximation for v, and should be refined. We argue that by combining these two criteria—edge weight difference between v and $\bar{v}$, which is a property of the current abstraction, together with the difference between $v(x)$ and $\bar{v}(x)$, which is a property of the specific input x, we can identify abstract neurons that have contributed significantly to x being a spurious counterexample.

The refinement heuristic is formally defined in Algorithm 3. The algorithm traverses the original neurons, looks for the edge weight times assignment value that has changed the most as a result of the current abstraction, and then performs refinement on the neuron at the end of that edge. As was the case with Algorithm 2, if considering all possible nodes turns out to be too costly, it is possible to adjust the algorithm to explore only some of the nodes, and pick the best one among those considered—without jeopardizing the algorithm's soundness.

Algorithm 3. Counterexample-Guided Refinement$(N, \bar{N}, x)$

```
 1: bestNeuron ← $\perp$, $m \leftarrow 0$
 2: for each concrete neuron $v_{i,j}$ of $N$ mapped into abstract neuron $\bar{v}_{i,j'}$ of $\bar{N}$ do
 3:    for each concrete neuron $v_{i-1,k}$ of $N$ mapped into abstract neuron $\bar{v}_{i-1,k'}$ of $\bar{N}$
       do
 4:       if $|w(v_{i-1,k}, v_{i,j}) - \bar{w}(\bar{v}_{i-1,k'}, \bar{v}_{i,j'})| \cdot |v_{i,j}(x) - \bar{v}_{i,j'}(x)| > m$ then
 5:          $m \leftarrow |w(v_{i-1,k}, v_{i,j}) - \bar{w}(\bar{v}_{i-1,k'}, \bar{v}_{i,j'})| \cdot |v_{i,j}(x) - \bar{v}_{i,j'}(x)|$
 6:          bestNeuron ← $v_{i,j}$
 7:       end if
 8:    end for
 9: end for
10: Use refine to split bestNeuron from its abstract neuron
```

As an example, let us use Algorithm 3 to choose a refinement step for the right hand side network of Fig. 5, for a spurious counterexample $\langle x_1, x_2 \rangle = \langle 1, 0 \rangle$. For this input, the original neurons' evaluation is $v_1 = 1, v_2 = 4$ and $v_3 = 2$, whereas the abstract neuron that represents them evaluates to 4. Suppose v_1 is considered first. In the abstract network, $\bar{w}(x_1, \bar{v_1}) = 4$ and $\bar{w}(x_2, \bar{v_1}) = -1$; whereas in the original network, $w(x_1, v_1) = 1$ and $w(x_2, v_1) = -2$. Thus, the largest value m computed for v_1 is $|w(x_1, v_1) - \bar{w}(x_1, \bar{v_1})| \cdot |4 - 1| = 3 \cdot 3 = 9$. This value of m is larger than the one computed for v_2 (0) and for v_3 (4), and so v_1 is selected for the refinement step. After this step is performed, v_2 and v_3 are still mapped to a single abstract neuron, whereas v_1 is mapped to a separate neuron in the abstract network.

5 Implementation and Evaluation

Our implementation of the abstraction-refinement framework includes modules that read a DNN in the NNet format [19] and a property to be verified, create an initial abstract DNN as described in Sect. 4, invoke a black-box verification engine, and perform refinement as described in Sect. 4. The process terminates when the underlying engine returns either `UNSAT`, or an assignment that is a true counterexample for the original network. For experimentation purposes, we integrated our framework with the Marabou DNN verification engine [22]. Our implementation and benchmarks are publicly available online [9].

Our experiments included verifying several properties of the 45 ACAS Xu DNNs for airborne collision avoidance [19,20]. ACAS Xu is a system designed to produce horizontal turning advisories for an unmanned aircraft (the *ownship*), with the purpose of preventing a collision with another nearby aircraft (the *intruder*). The ACAS Xu system receive as input sensor readings, indicating the location of the intruder relative to the ownship, the speeds of the two aircraft, and their directions (see Fig. 6). Based on these readings, it selects one of 45 DNNs, to which the readings are then passed as input. The selected DNN then assigns scores to five output neurons, each representing a possible turning advisory: strong left, weak left, strong right, weak right, or clear-of-conflict (the latter indicating that it is safe to continue along the current trajectory). The neuron with the *lowest* score represents the selected advisory. We verified several properties of these DNNs based on the list of properties that appeared in [20]—specifically focusing on properties that ensure that the DNNs always advise clear-of-conflict for distant intruders, and that they are robust to (i.e., do not change their advisories in the presence of) small input perturbations.

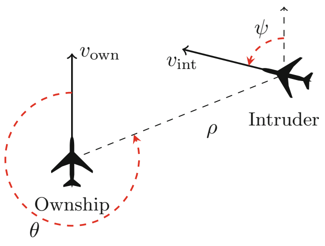


Fig. 6. (From [20]) An illustration of the sensor readings passed as input to the ACAS Xu DNNs.

Each of the ACAS Xu DNNs has 300 hidden nodes spread across 6 hidden layers, leading to 1200 neurons when the transformation from Sect. 3.1 is applied. In our experiments we set out to check whether the abstraction-based approach could indeed prove properties of the ACAS Xu networks on abstract networks that had significantly fewer neurons than the original ones. In addition, we wished to compare the proposed approaches for generating initial abstractions (the abstraction to saturation approach versus the indicator-guided abstraction described in Algorithm 2), in order to identify an optimal configuration for our tool. Finally, once the optimal configuration has been identified, we used it to compare our tool's performance to that of vanilla Marabou. The results are described next.

Figure 7 depicts a comparison of the two approaches for generating initial abstractions: the abstraction to saturation scheme (x axis), and the indicator-guided abstraction scheme described in Algorithm 2 (y axis). Each experiment included running our tool twice on the same benchmark (network and property), with an identical configuration except for the initial abstraction being used. The plot depicts the total time (log-scale, in seconds, with a 20-h timeout) spent by Marabou solving verification queries as part of the abstraction-refinement procedure. It shows that, in contrast to our intuition, abstraction to saturation almost always outperforms the indicator-guided approach. This is perhaps due to the fact that, although it might entail additional rounds of refinement, the abstraction to saturation approach tends to produce coarse verification queries that are easily solved by Marabou, resulting in an overall improved performance. We thus conclude that, at least in the ACAS Xu case, the abstraction to saturation approach is superior to that of indicator-guided abstraction.

This experiment also confirms that properties can indeed be proved on abstract networks that are significantly smaller than the original—i.e., despite the initial 4x increase in network size due to the preprocessing phase, the final abstract network on which our abstraction-enhanced approach could solve the query was usually substantially smaller than the original network. Specifically, among the abstraction to saturation experiments that terminated, the final network on which the property was shown to be `SAT` or `UNSAT` had an average size of 268.8 nodes, compared to the original 310—a 13% reduction. Because DNN verification becomes exponentially more difficult as the network size increases, this reduction is highly beneficial.

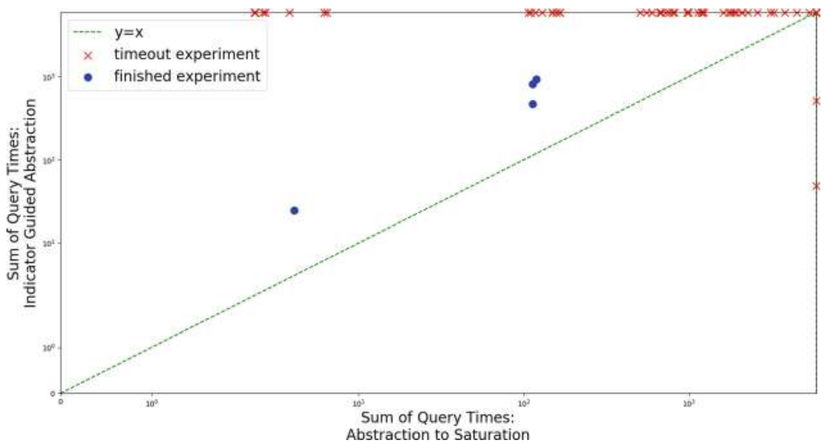


Fig. 7. Generating initial abstractions using abstraction to saturation and indicator-guided abstraction.

Next, we compared our abstraction-enhanced Marabou (in abstraction to saturation mode) to the vanilla version. The plot in Fig. 8 compares the total query solving time of vanilla Marabou (y axis) to that of our approach (x axis). We ran the tools on 90 ACAS Xu benchmarks (2 properties, checked on each of the 45 networks), with a 20-h timeout. We observe that the abstraction-enhanced version significantly outperforms vanilla Marabou on average—often solving queries orders-of-magnitude more quickly, and timing out on fewer benchmarks. Specifically, the abstraction-enhanced version solved 58 instances, versus 35 solved by Marabou. Further, over the instances solved by both tools, the abstraction-enhanced version had a total query median runtime of 1045 s, versus 63671 s for Marabou. Interestingly, the average size of the abstract networks for which our tool was able to solve the query was 385 nodes—which is an increase compared to the original 310 nodes. However, the improved runtimes demonstrate that although these networks were slightly larger, they were still much easier to verify, presumably because many of the network's original neurons remained abstracted away.

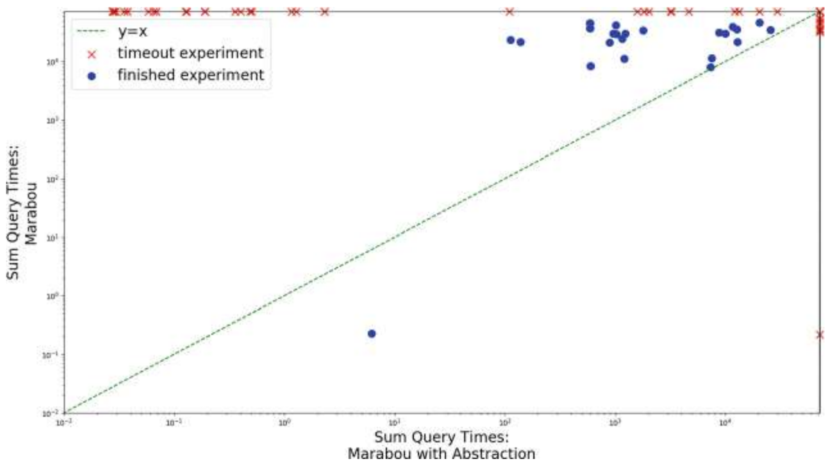


Fig. 8. Comparing the run time (in seconds, logscale) of vanilla Marabou and the abstraction-enhanced version on the ACAS Xu benchmarks.

Finally, we used our abstraction-enhanced Marabou to verify *adversarial robustness* properties [35]. Intuitively, an adversarial robustness property states that slight input perturbations cannot cause sudden spikes in the network's output. This is desirable because such sudden spikes can lead to misclassification of inputs. Unlike the ACAS Xu domain-specific properties [20], whose formulation required input from human experts, adversarial robustness is a *universal property*, desirable for every DNN. Consequently it is easier to formulate, and has received much attention (e.g., [2,10,20,36]).

In order to formulate adversarial robustness properties for the ACAS Xu networks, we randomly sampled the ACAS Xu DNNs to identify input points where the selected output advisory, indicated by an output neuron y_i, received a much lower score than the second-best advisory, y_j (recall that the advisory with the lowest score is selected). For such an input point x_0, we then posed the verification query: does there exist a point x that is close to x_0, but for which y_j receives a lower score than y_i? Or, more formally: $(\|x - x_0\|_{L_\infty} \leq \delta) \wedge (y_j \leq y_i)$. If this query is `SAT` then there exists an input x whose distance to x_0 is at most δ, but for which the network assigns a better (lower) score to advisory y_j than to y_i. However, if this query is `UNSAT`, no such point x exists. Because we select point x_0 such that y_i is initially much smaller than y_j, we expect the query to be `UNSAT` for small values of δ.

For each of the 45 ACAS Xu networks, we created robustness queries for 20 distinct input points—producing a total of 900 verification queries (we arbitrarily set $\delta = 0.1$). For each of these queries we compared the runtime of vanilla Marabou to that of our abstraction-enhanced version (with a 20-h timeout). The results are depicted in Fig. 9. Vanilla Marabou was able to solve more instances—893 out of 900, versus 805 that the abstraction-enhanced version was able to solve. However, on the vast majority of the remaining experiments, the abstraction-enhanced version was significantly faster, with a total query median runtime of only 0.026 s versus 15.07 s in the vanilla version (over the 805 benchmarks solved by both tools). This impressive 99% improvement in performance highlights the usefulness of our approach also in the context of adversarial robustness. In addition, over the solved benchmarks, the average size of the abstract networks for which our tool was able to solve the query was 104.4 nodes, versus 310 nodes in each of the original networks—a 66% reduction in size. This reinforces our statement that, in many cases, DNNs contain a great deal of unneeded neurons, which can safely be removed by the abstraction process for the purpose of verification.

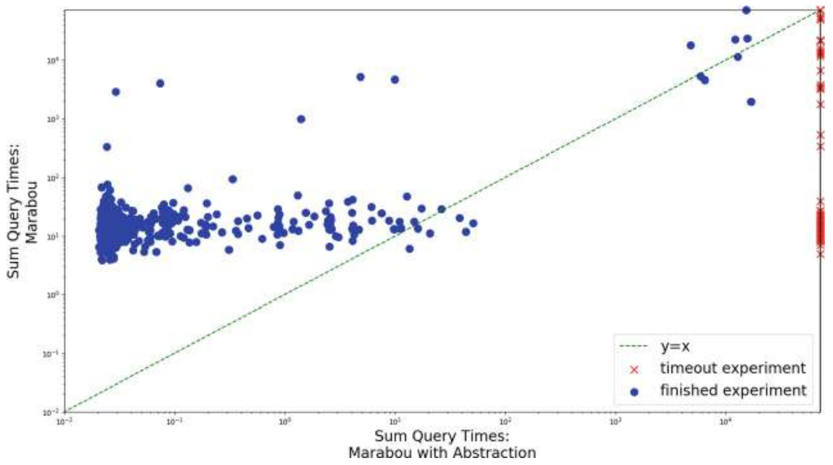


Fig. 9. Comparing the run time (seconds, logscale) of vanilla Marabou and the abstraction-enhanced version on the ACAS Xu adversarial robustness properties.

6 Related Work

In recent years, multiple schemes have been proposed for the verification of neural networks. These include SMT-based approaches, such as Marabou [22,23], Reluplex [20], DLV [17] and others; approaches based on formulating the problem as a mixed integer linear programming instance (e.g., [4,7,8,36]); approaches that use sophisticated symbolic interval propagation [37], or abstract interpretation [10]; and others (e.g., [1,18,25,27,30,38,39]). These approaches have been applied in a variety of tasks, such as measuring adversarial robustness [2,17], neural network simplification [11], neural network modification [12], and many others (e.g., [23,34]). Our approach can be integrated with any sound and complete solver as its engine, and then applied towards any of the aforementioned tasks. Incomplete solvers could also be used and might afford better performance, but this could result in our approach also becoming incomplete.

Some existing DNN verification techniques incorporate abstraction elements. In [31], the authors use abstraction to over-approximate the Sigmoid activation function with a collection of rectangles. If the abstract verification query they produce is UNSAT, then so is the original. When a spurious counterexample is found, an arbitrary refinement step is performed. The authors report limited scalability, tackling only networks with a few dozen neurons. Abstraction techniques also appear in the AI2 approach [10], but there it is the input property and reachable regions that are over-approximated, as opposed to the DNN itself. Combining this kind of input-focused abstraction with our network-focused abstraction is an interesting avenue for future work.

7 Conclusion

With deep neural networks becoming widespread and with their forthcoming integration into safety-critical systems, there is an urgent need for scalable techniques to verify and reason about them. However, the size of these networks poses a serious challenge. Abstraction-based techniques can mitigate this difficulty, by replacing networks with smaller versions thereof to be verified, without compromising the soundness of the verification procedure. The abstraction-based approach we have proposed here can provide a significant reduction in network size, thus boosting the performance of existing verification technology.

In the future, we plan to continue this work along several axes. First, we intend to investigate refinement heuristics that can split an abstract neuron into two arbitrary sized neurons. In addition, we will investigate abstraction schemes for networks that use additional activation functions, beyond ReLUs. Finally, we plan to make our abstraction scheme parallelizable, allowing users to use multiple worker nodes to explore different combinations of abstraction and refinement steps, hopefully leading to faster convergence.

Acknowledgements. We thank the anonymous reviewers for their insightful comments. This project was partially supported by grants from the Binational Science Foundation (2017662) and the Israel Science Foundation (683/18).

References

1. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 731–744 (2019)
2. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. In: Proceedings 30th Conference on Neural Information Processing Systems (NIPS) (2016)
3. Bojarski, M., et al.: End to end learning for self-driving cars. Technical report (2016). http://arxiv.org/abs/1604.07316
4. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Kumar, M.: Piecewise linear neural network verification: a comparative study. Technical report (2017). https://arxiv.org/abs/1711.00455v1
5. Carlini, N., Katz, G., Barrett, C., Dill, D.: Provably minimally-distorted adversarial examples. Technical report (2017). https://arxiv.org/abs/1709.10207
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings 12th Internation Conference on Computer Aided Verification (CAV), pp. 154–169 (2010)
7. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks. In: Proceedings 10th NASA Formal Methods Symposium (NFM), pp. 121–138 (2018)
8. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings 15th Internatioanl Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 269–286 (2017)
9. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification: proof-of-concept implementation (2020). https://drive.google.com/file/d/1KCh0vOgcOR2pSbGRdbtAQTmoMHAFC2Vs/view
10. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings 39th IEEE Symposium on Security and Privacy (S&P) (2018)
11. Gokulanathan, S., Feldsher, A., Malca, A., Barrett, C., Katz, G.: Simplifying neural networks using formal verification. In: Proceedings 12th NASA Formal Methods Symposium (NFM) (2020)
12. Goldberger, B., Adi, Y., Keshet, J., Katz, G.: Minimal modifications of deep neural networks using verification. In: Proceedings 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) (2020)
13. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
14. Gopinath, D., Katz, G., Păsăreanu, C.S., Barrett, C.: DeepSafe: a data-driven approach for assessing robustness of neural networks. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 3–19. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_1
15. Gottschlich, J., et al.: The three pillars of machine programming. In: Proceedings 2nd ACM SIGPLAN Internatioanl Workshop on Machine Learning and Programming Languages (MALP), pp. 69–80 (2018)
16. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Proces. Mag. **29**(6), 82–97 (2012)

17. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
18. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. Technical report (2020). http://arxiv.org/abs/2004.02462
19. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings 35th Digital Avionics Systems Conference (DASC), pp. 1–10 (2016)
20. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
21. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Towards proving the adversarial robustness of deep neural networks. In: Proceedings 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV), pp. 19–26 (2017)
22. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
23. Kazak, Y., Barrett, C., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: Proceedings 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI) (2019)
24. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
25. Kuper, L., Katz, G., Gottschlich, J., Julian, K., Barrett, C., Kochenderfer, M.: Toward scalable verification for safety-critical deep networks. Technical report (2018). https://arxiv.org/abs/1801.05950
26. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. Technical report (2016). http://arxiv.org/abs/1607.02533
27. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. Technical report (2017). https://arxiv.org/abs/1706.07351
28. Mao, H., Netravali, R., Alizadeh, M.: Neural adaptive video streaming with Pensieve. In: Proceedings Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), pp. 197–210 (2017)
29. Nair, V., Hinton, G.: Rectified linear units improve restricted boltzmann machines. In: Proceedings 27th International Conference on Machine Learning (ICML), pp. 807–814 (2010)
30. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. Technical report (2017). http://arxiv.org/abs/1709.06662
31. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
32. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: Proceedings 27th International Joint Conference on Artificial Intelligence (IJACI), pp. 2651–2659 (2018)
33. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)

34. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2019)
35. Szegedy, C., et al.: Intriguing properties of neural networks. Technical report (2013). http://arxiv.org/abs/1312.6199
36. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: Proceedings 7th International Conference on Learning Representations (ICLR) (2019)
37. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings 27th USENIX Security Symposium (2018)
38. Wu, H., et al.: Parallelization techniques for verifying neural networks. Technical report (2020). https://arxiv.org/abs/2004.08440
39. Xiang, W., Tran, H.-D., Johnson, T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Networks Learn. Syst. (TNNLS) **99**, 1–7 (2018)

Improved Geometric Path Enumeration for Verifying ReLU Neural Networks

Stanley Bak[1](✉), Hoang-Dung Tran[2,3], Kerianne Hobbs[4,5], and Taylor T. Johnson[3]

[1] Stony Brook University, Stony Brook, USA
stanleybak@gmail.com
[2] University of Nebraska, Lincoln, USA
[3] Vanderbilt University, Nashville, USA
[4] Air Force Research Laboratory, Wright-Patterson Air Force Base, USA
[5] Georgia Institute of Technology, Atlanta, USA

Abstract. Neural networks provide quick approximations to complex functions, and have been increasingly used in perception as well as control tasks. For use in mission-critical and safety-critical applications, however, it is important to be able to analyze what a neural network can and cannot do. For feed-forward neural networks with ReLU activation functions, although exact analysis is NP-complete, recently-proposed verification methods can sometimes succeed.

The main practical problem with neural network verification is excessive analysis runtime. Even on small networks, tools that are theoretically complete can sometimes run for days without producing a result. In this paper, we work to address the runtime problem by improving upon a recently-proposed geometric path enumeration method. Through a series of optimizations, several of which are new algorithmic improvements, we demonstrate significant speed improvement of exact analysis on the well-studied ACAS Xu benchmarks, sometimes hundreds of times faster than the original implementation. On more difficult benchmark instances, our optimized approach is often the fastest, even outperforming inexact methods that leverage overapproximation and refinement.

1 Introduction

Neural networks have surged in popularity due to their ability to learn complex function approximations from data. This ability has led to their proposed application in perception and control decision systems, which are sometimes safety-critical. For use in safety-critical applications, it is important to prove properties about neural networks rather than treating them as black-box components.

DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA #88ABW-2020-0116, 15 JAN 2020).

A recent method [24] based on path enumeration and geometric set propagation has shown that exact analysis can be practical for piecewise linear neural networks. This includes networks with fully-connected layers, convolutional layers, average and max pooling layers, and neurons with ReLU activation functions. Here, we focus on fully-connected layers with ReLU activation functions. The verification problem in this method is presented in terms of input/output properties of the neural network. The method works by taking the input set of states and performing a set-based execution of the neural network. Due to the linear nature of the set representation and the piecewise linear nature of the ReLU activation function, the set may need to be split after each neuron is executed, so that the output after the final layer is a collection of sets that can each be checked for intersection with an unsafe set.

Since the formal verification problem we are addressing has been shown to be NP-Complete [13], we instead focus on improving practical scalability. This requires us to choose a set of benchmarks for evaluation. For this, we focus on properties from the well-studied ACAS Xu system [13]. This contains a mix of safe and unsafe instances, where the original verification times measured from seconds to days, including some unsolved instances.

The main contributions of this paper are:

- several new speed improvements to the path enumeration method, along with correctness justifications, that are each systematically evaluated;
- the first verification method that verifies all 180 benchmark instances from ACAS Xu properties 1–4, each in under 10 min on a standard laptop;
- a comparison with other recent tools, including Marabou, Neurify, NNV, and ERAN, where our method is often the fastest and over 100x faster than the original path enumeration method implementation in NNV.

This paper first reviews background related to neural networks, the path enumeration verification approach, and the ACAS Xu benchmarks in Sect. 2. Next, Sect. 3 analyzes several algorithmic optimizations to the basic procedure, and systematically evaluates each optimization's effect on the execution times of the ACAS Xu benchmarks. A comparison with other tools is provided in Sect. 4, followed by review of related work in Sect. 5 and a conclusion.

2 Background

We now review the neural network verification problem (Sect. 2.1), the basic geometric path enumeration algorithm (Sect. 2.2), important spatial data structures (Sect. 2.3), and the ACAS Xu benchmarks (Sect. 2.4).

2.1 Neural Networks and Verification

In this work, we focus our attention on fully-connected, feedforward neural networks with ReLU activation functions. A neural network computes a function $\mathsf{NN} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_o}$, where n_i is the number of inputs and n_o is the number of outputs. A neural network consists of k layers, where each layer i is defined with a weight matrix W_i and a bias vector b_i. Given an input point $y_0 \in \mathbb{R}^{n_i}$, a neural network will compute an output point $y_k \in \mathbb{R}^{n_o}$ as follows:

$$\begin{aligned} x^{(1)} &= W_1 y_0 + b_1, \quad & y_1 &= f(x^{(1)}) \\ x^{(2)} &= W_2 y_1 + b_2, \quad & y_2 &= f(x^{(2)}) \\ &\vdots \\ x^{(k)} &= W_k y_{k-1} + b_k, \quad & y_k &= f(x^{(k)}) \end{aligned}$$

We call y_{i-1} and y_i the input and output of the i-th layer, respectively, and $x^{(i)}$ the intermediate values at layer i. The vector-function f is defined using a so-called *activation function*, that is applied element-wise to the vector of intermediate values at each layer. We focus on the popular rectified linear unit (ReLU) activation function, $\mathsf{ReLU}(x) = \max(x, 0)$.

For this computation definition to make sense, the sizes of the weights matrices and bias vectors are restricted. The first layer must accept n_i-dimensional inputs, the final layer must produce n_o-dimensional outputs, and the intermediate layers must have weights and biases that have sizes compatible with their immediate neighbors, in the sense of matrix/vector multiplication and addition. The number of neurons (sometimes called hidden units) at layer i is defined as the number of elements in the layer's output vector y_i.

Definition 1 (Output Range). *Given a neural network that computes the function* NN *and an input set* $\mathcal{I} \subseteq \mathbb{R}^{n_i}$*, the* ***output range*** *is the set of possible outputs of the network, when executed from a point inside the input set,* $\mathsf{Range}(\mathsf{NN}, \mathcal{I}) = \{y_k \mid y_k = \mathsf{NN}(y_0),\ y_0 \in \mathcal{I}\}$.

Computing the output range is one way to solve the verification problem.

Definition 2 (Verification Problem for Neural Networks). *Given a neural network that computes the function* NN*, an input set* $\mathcal{I} \subseteq \mathbb{R}^{n_i}$*, and an unsafe set* $\mathcal{U} \subseteq \mathbb{R}^{n_o}$*, the* ***verification problem for neural networks*** *is to check if* $\mathsf{Range}(\mathsf{NN}, \mathcal{I}) \cap \mathcal{U} = \emptyset$.

If verification is impossible, we would also prefer to generate a counterexample $y_0 \in \mathcal{I}$ where $y_k = \mathsf{NN}(y_0)$ and $y_k \in \mathcal{U}$, although not all tools do this. We also further assume in this work that the input and unsafe sets are defined with linear constraints, $\mathcal{I} = \{x \mid A_i x \leq b_i, x \in \mathbb{R}^{n_i}\}$, and $\mathcal{U} = \{x \mid A_u x \leq b_u, x \in \mathbb{R}^{n_o}\}$.

2.2 Basic Geometric Path Enumeration Algorithm

Given enough time, the output range of a neural network can be computed exactly using a recently-proposed geometric path enumeration approach [24].

```
input  : Input Set: $\mathcal{I}$, Unsafe Set: $\mathcal{U}$
output: Verification Result (safe or unsafe)
1 $s \leftarrow \langle$layer:0, neuron:None, $\theta$ : convert($\mathcal{I}$)$\rangle$ // computation-state tuple
2 $\mathcal{W} \leftarrow$ List() // initialize waiting list
3 $\mathcal{W}$.put($s$)
4 result $\leftarrow$ safe
5 while result = safe and $\neg\mathcal{W}$.empty() do
6     $s \leftarrow \mathcal{W}$.pop()
7     result $\leftarrow$ step($s$,$\mathcal{W}$,$\mathcal{U}$) // updates $\mathcal{W}$, given in Algorithm 2
8 end
9 return result
```

Algorithm 1: High-level neural-network path enumeration algorithm.

The general strategy is to execute the neural network with *sets* instead of points. A *spatial data structure* is used to represent the input set of states, and this set is propagated through each layer of the neural network, computing the set of possible intermediate values and then the set of possible outputs repeatedly until the output of the final layer is computed. In this context, a spatial data structure represents some subset of states in a Euclidean space $\mathbb{R}^n$, where the number of dimensions n is the number of neurons in one of the layers of the network, and may change as the set is propagated layer by layer. An example spatial data structure could be a polytope defined using a finite set of half-spaces (linear constraints), although as explained later this is not the most efficient choice. Section 2.3 will discuss spatial data structures in more detail.

The high-level verification method is shown in Algorithm 1, where functions in red are custom to the spatial data structure being used. The `convert` function (line 1) converts the input set $\mathcal{I}$ from linear constraints to the desired spatial data structure, and stores it in the θ element of s, where s is called a *computation-state tuple*. A `neuron` value of None in the tuple indicates that next operation should be an affine transformation. The computation-state tuple is then put into a waiting list (line 3), which stores tuples that need further processing. The `step` function (line 7) propagates the set θ by a single neuron in a single layer of the network, and is elaborated on in the next paragraph. This function can modify $\mathcal{W}$, possibly inserting one or more computation-state tuples, although always at a point further along in the network (with a larger layer number or neuron index), which ensures eventual termination of the loop. This function will also check if the set, after being fully propagated through the network, intersects the unsafe set. In this case, `step` will return unsafe, which causes the `while` loop to immediately terminate since the result is known.

The `step` function propagates the set of states θ by one neuron, and is shown in Algorithm 2. The intermediate values are computed from the input set of each layer by calling `affine_transformation` (line 12). For the current neuron index n, the algorithm will check if the input to the ReLU activation function, dimension n of the set θ, is always positive (or zero), always negative, or can be either positive or negative. This is done by the `get_sign` function (line 21), which

```
input  : Computation-State Tuple: $s$, Waiting List: $\mathcal{W}$, Unsafe Set: $\mathcal{U}$
output: Safe so far? (safe or unsafe)
if $s$.neuron = None then
    // finished with the previous layer
    if $s$.layer = $k$ then
        // finished with all layers
        if $s.\theta$.has_intersection($\mathcal{U}$) = $\emptyset$ then
            return safe
        else
            return unsafe // alternatively, return counterexample here
        end
    else
        $s$.layer ← $s$.layer + 1
        $s.\theta$.affine_transformation($W_{s.\text{layer}}, b_{s.\text{layer}}$)
        $s$.neuron ← 1
    end
end
$n$ ← $s$.neuron
$s$.neuron ← $n + 1$
if $s$.neuron > size($b_{s.\text{layer}}$) then
    $s$.neuron ← None // $n$ is the last neuron in the current layer
end
switch get_sign($s, n$) do
    case pos do
        // do nothing
    case neg do
        $s.\theta$.project_to_zero($n$)
    case posneg do
        $t$ ← ⟨$s$.layer, $s$.neuron, $s.\theta$⟩ // deep copy $s$
        $s.\theta$.add_constraint($n, \geq, 0$) // split on positive case
        $t.\theta$.add_constraint($n, \leq, 0$) // split on negative case
        $t.\theta$.project_to_zero($n$)
        $\mathcal{W}$.put($t$)
end
$\mathcal{W}$.put($s$)
return safe // safe so far
```

Algorithm 2: Pseudocode for `step` function, which propagates a set through the network by one neuron.

returns pos, neg, or posneg, respectively. In the first two cases, the current dimension n of the set is left alone or assigned to zero (using the `project_to_zero` method), to reflect the semantics of the ReLU activation function when the input is positive or negative, respectively. In the third case, the set is split into two sets along linear constraint where the input to the activation function equals zero. In the case where the input to the activation function is less than zero, the value of dimension n is projected to zero, reflecting the semantics of the ReLU activation function. The splitting is done using the `add_constraint` method of the spatial data structure, which takes three arguments: n, `sign`, and `val`. This method intersects the set with the linear condition that the n-th dimension is, depending on `sign`, greater than, less than, and/or equal to `val`. Once the set has been propagated through the whole network, it is checked for intersection with the unsafe set (line 5), using the `has_intersection` method.

This enumeration algorithm has been shown to be sound and complete [24]. However, for this strategy to work in practice, the spatial data structure used to store θ must support certain operations *efficiently*. These are denoted in red in Algorithms 1 and 2: `convert`, `has_intersection`, `affine_transformation`, `get_sign`, `project_to_zero`, and `add_constraint`. Polytopes represented with half-spaces, for example, do not have a known efficient way to compute general affine transformations in high dimensions. Instead, linear star sets [4] will be used, which are a spatial data structure that support all the required operations efficiently and without overapproximation error. These will be elaborated on more in the next subsection.

In this work, we focus on optimizations to the presented algorithm that increase its practical scalability, while exploring the same set of paths. The most important factor that we do not control and influences whether this can succeed is the number of paths that exist. Each output set that gets checked for intersection with the unsafe set corresponds to a unique *path* through the network, where the path is defined by the sign of each element of the intermediate values vector at each layer. The algorithm enumerates every path of the network for a given input set. An upper bound on this is 2^N, where N is the total number of neurons in all the layers of the network. For many practical verification problem instances, however, the actual number of unique paths is significantly smaller than the upper bound.

2.3 Spatial Data Structures

Using the correct spatial data structure (set representation in this context) is important to the efficiency of Algorithm 1 and 2, as well as some of our optimizations. Here we review two important spatial data structures, zonotopes and (linear) star sets.

Zonotopes. A *zonotope* is an affine transformation of the $[-1, 1]^p$ box. Zonotopes have been used for efficient analysis of hybrid systems [8] as well as more recently to verify neural networks using overapproximations [7,21]. Zonotopes can be described mathematically as $Z = (c, G)$, where the *center* c is an n-dimensional vector and *generator matrix* G is an $n \times p$ matrix. The columns of G are sometimes referred to as *generators* of the zonotope, and we write these as $g_1, \ldots, g_p$. A zonotope Z encodes a set of states as:

$$Z = \left\{x \in \mathbb{R}^n \;\middle|\; x = c + G\alpha,\ \alpha \in [-1, 1]^p\right\} \tag{1}$$

The two most important properties of zonotopes for the purposes of verification are that they are efficient for (i) affine transformation, and (ii) optimization.

An affine transformation of an n-dimensional point x to a q-dimensional space is defined with a $q \times n$ matrix A and q-dimensional vector b so that the transformed point is $x' = Ax+b$. An affine transformation of every point in an n-dimensional set of points described by a zonotope $Z = (c, G)$ is easily computed as $Z' = (Ac + b, AG)$. Note this uses standard matrix operations which scale polynomially with the dimension of A, and are especially efficient if the number of generators is small. In the verification problem, the number of generators, p, corresponds to the degrees of freedom needed to encode the input set of states. In ACAS Xu system, for example, there are 5 inputs, and so the input set can be encoded with 5 generators. In contrast, affine transformations of polytopes require converting between a half-space and vertex representation, which is slow.

The second efficient operation for zonotopes is optimization in some direction vector v. Given a zonotope $Z = (c, G)$ and a direction v to maximize, the point $x^* \in Z$ that maximizes the dot product $v \cdot x^*$ can be obtained as a simple summation $x^* = c + \sum_{i=1}^{p} x_i^*$, where each x_i^* is given as:

$$x_i^* = \begin{cases} v_i, & \text{if } v_i \cdot g_i \geq -v_i \cdot g_i \\ -v_i, & \text{otherwise} \end{cases} \tag{2}$$

Star Sets. A (linear) *star set* is another spatial data structure that generalizes a zonotope. A star set is an affine transformation of an arbitrary p-dimensional polytope. Mathematically, a star set S is a 3-tuple, (c, G, P), where c and G are the same as with a zonotope, and P is a half-space polytope in p dimensions. A star set S encodes a set of states (compare with Eq. 1):

$$S = \{x \in \mathbb{R}^n \mid x = c + G\alpha,\ \alpha \in P\} \tag{3}$$

A star set can encode any zonotope by letting P be the $[-1, 1]^p$ box. Star sets can also encode more general sets than zonotopes by using a more complex polytope P. A triangle, for example, can be encoded as a star set by setting P to be a triangle, using the origin as c and the identity matrix as V. This cannot be encoded with zonotopes, as they must be centrally symmetric. In Algorithm 1 on line 1, the `convert` function produces the input star set (c, G, P) from input polytope $\mathcal{I}$ setting c to the zero vector, G to the identity matrix, and P to $\mathcal{I}$.

Affine transformations by a $q \times n$ matrix A and q-dimensional vector b of a star set S can be computed efficiently similar to a zonotope: $S' = (Ac+b, AG, P)$.

Optimization in some direction v is slightly less efficient than with a zonotope, and can be done using linear programming (LP). To find a point $x^* \in S$ that maximizes the dot product $v \cdot x^*$, we convert the optimization direction v to the initial space $w = (vG)^T$, find a point $\alpha^* \in P$ that maximizes w using LP, and then convert α^* back to the n-dimensional space $x^* = c + G\alpha^*$.

Star sets, unlike zonotopes, also efficiently support half-space intersection operations by adding constraints to the star set's polytope. Given a star set $S = (c, G, P)$ and an n-dimensional half-space $dx \leq \mathtt{e}$ defined by vector d and scalar $\mathtt{e}$, we convert this to a p-dimensional half-space as follows:

$$(dG)\alpha \leq \mathtt{e} - dc \tag{4}$$

The star set after intersection is then $S' = (c, G, P')$, where the half-space polytope P' is the same as P, with one additional constraint given by Eq. 4.

2.4 ACAS Xu Benchmarks

Since the verification problem for neural networks is NP-Complete, we know exact analysis methods cannot work well in all instances. In order to evaluate improvements, therefore, we must focus on a set of benchmarks.

In this work, we choose to focus on the Airborne Collision System X Unmanned (ACAS Xu) set of neural network verification benchmarks [13]. As these benchmarks have been widely-used for evaluation in other publications, and some authors have even made their tools available publicly, using these allows us to provide a common comparison point with other methods later in Sect. 4.

ACAS Xu is a flight-tested aircraft system designed to avoid midair collisions of unmanned aircraft by issuing horizontal maneuver advisories [17]. The system was designed using a partially observable Markov decision process that resulted in a 2 GB lookup table which mapped states to commands. This mapping was compressed to 3 MB using 45 neural networks (two of the inputs were discretized and are used to choose the applicable network) [12]. Since the compression is not exact, the verification step checks if the system still functions correctly.

Each network contains five inputs that get set to the current the aircraft state, and five outputs that determine the current advisory. The network has six ReLU layers with 50 neurons each, for a total of 300 neurons. Ten properties were originally defined, encoding things like, if the aircraft are approaching each other head-on, a turn command will be advised (property 3). The formal definition of all the properties encoded as linear constraints is available in the appendix of the original work [13].

3 Improvements

We now systematically explore several improvements to the exact path enumeration verification method from Sect. 2.2. For each proposed improvement, we compare the run-time on the ACAS Xu system with and without the change. We focus on properties 1–4. Although originally these were measured on a subset of the 45 networks [13], the same authors later used all the networks to check these properties [14], which is what we will do here. Each verification instance is run with a 10 min timeout, so that the maximum time needed to test a single method, if a timeout is encountered on each of the 180 benchmarks, is 30 h. Later, in Sect. 4, we will compare the most optimized method with other verification tools and the other ACAS Xu properties. Unless indicated otherwise, our experiments were performed on a Laptop platform with Ubuntu Linux 18.04, 32 GB RAM and an Intel Xeon E-2176M CPU running at 2.7 GHz with 6 physical cores (12 virtual cores with hyperthreading). The full data measurements summarized in this section are provided in Appendix C.

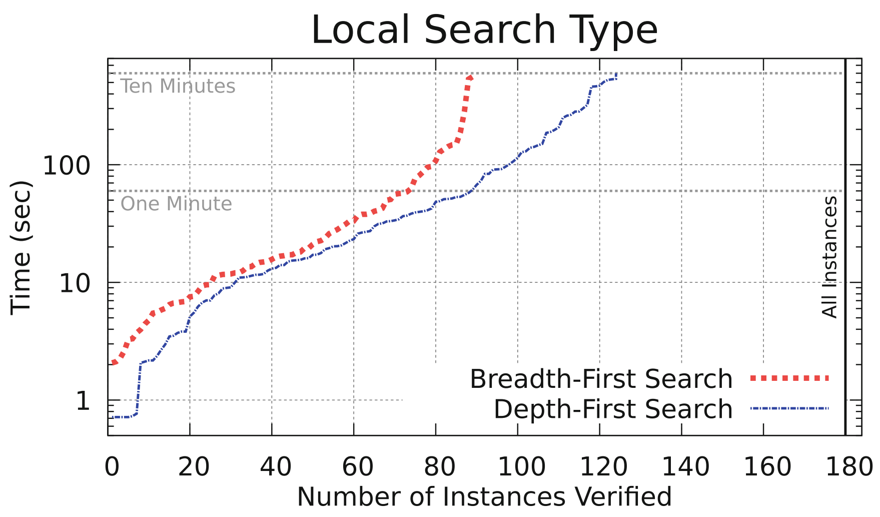


Fig. 1. Depth-first search outperforms breadth-first search.

3.1 Local Search Type (DFS vs BFS)

Algorithm 1 uses a waiting list to store the computation-state tuples, which are popped off one at a time and passed to the `step` function. This need not strictly be a list, but is rather a collection of computation-state tuples, and we can consider changing the order states are popped to explore the state space with different strategies. If the possible paths through the neural network are viewed as a tree, two well-known strategies for tree traversal that can be considered are depth-first search (DFS) and breadth-first search (BFS). A DFS search can be performed popping the computation-state tuple with the largest $(\mathsf{layer}, \mathsf{neuron})$ pair, whereas a BFS search is done by popping the tuple with the smallest $(\mathsf{layer}, \mathsf{neuron})$ pair.

The original path enumeration with star set approach [24] describes a layer-by-layer exploration strategy, which is closer to a BFS search. Finite-state machine model-checking methods, however, more often use DFS search.

We compare the two approaches in Fig. 1, which summarizes the execution of all 180 benchmarks. Here, the y-axis is a timeout in seconds, and the x-axis is the number of benchmarks verified within that time. Within the ten minute timeout, around 90 benchmarks can be successfully verified with BFS, and 120 with DFS[1]. Notice that the y-axis is log scale, so that differences in runtimes between easy and hard benchmark instances are both visible.

As can be seen in the figure, the DFS strategy is superior. This is primarily due to unsafe instances of the benchmarks, where DFS can often quickly find an unsafe execution and exit the high-level loop, whereas BFS first iterates through

[1] The DFS method solves every benchmark that can be solved with BFS. Appendix C contains the complete results.

all the layers and neurons (DFS explores deep paths, which sometimes are quickly found to be unsafe). In the cases where the system was safe, both approaches took similar time. Another known advantage of DFS search is that the memory needed to store the waiting list is significantly smaller, which can be a factor for the benchmarks with a large number of paths.

Correctness Justification: Both DFS and BFS explore the same sets of states, just in a different order.

3.2 Bounds for Splitting

Using DFS search, we consider other improvements. The original path enumeration publication mentions the following optimization:

> "... to minimize the number of [operations] and computation time, we first determine the ranges of all states in the input set which can be done efficiently by solving ... linear programming problems." [24]

An evaluation of the improvement is not provided, so we investigate this here. The optimization is referring to the implementation of the `get_sign` function on line 21 of Algorithm 2. The `get_sign`(s, n) function takes as input a computation-state tuple s with spatial data structure θ (a star set) and a dimension number n. It returns pos, neg, or posneg, depending on whether value of dimension n, which we call x_n, in set θ can be positive (or zero), negative or both. Our baseline implementation, which we refer to as Copy, determines the output of `get_sign` by creating two copies of the passed-in star set, intersecting them with the condition that $x_n \leq 0$ or $x_n \geq 0$, and then checking each star set for feasibility, done using linear programming (LP). In the second version, which we call Bounds, the passed-in star set is instead minimized and maximized in the direction of x_n, to determine the possible signs. While Copy incurs overhead from creating copies and adding intersections, Bounds does extra work by computing the minimum and maximum which are not really needed (we only need the possible signs of x_n).

A comparison of the optimizations on the ACAS Xu benchmarks are shown in Fig. 2 by comparing Copy to Bounds, we confirm the original paper's claim that Bounds is faster.

Correctness Justification: If θ intersected with $x_n \leq 0$ is feasible, then the minimum value of x_n in θ will be less than or equal to zero and vice versa. Similar for the maximum case.

3.3 Fewer LPs with Concrete Simulations

We next consider strategies to determine the possible signs of a neuron's output with fewer LP calls, which we call *prefiltering*. Consider a modification of the Bounds optimization, where rather than computing both the upper and lower bound of x_n, we first compute the lower bound and check if its value is positive.

If this is the case, we know get_sign should return pos, and we do not need to compute the upper bound. We could, alternatively, first compute the upper bound and check if its value is negative. If there is no branching and we guess the correct side to check, only a single LP needs to be solved instead of two.

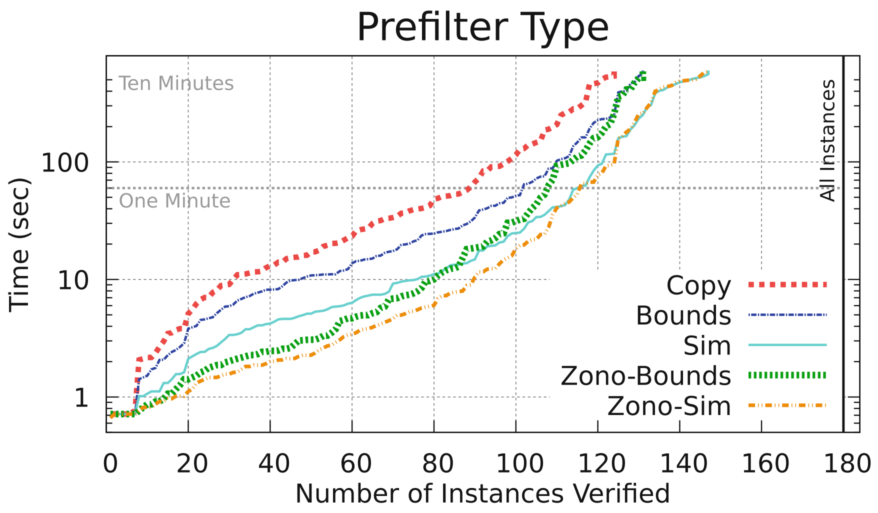


Fig. 2. Prefilter optimizations improve performance by rejecting branches without LP solving. The Zono-Sim method works best.

We can do even better than guessing by tracking extra information in the computation-state tuple. We add a `simulation` field to s, which contains a concrete value in the set of states θ. This is initialized to any point in the input set $\mathcal{I}$, which can be obtained using LP, or using the center point if the input states are a box. When get_sign returns posneg and the set is split (line 27 in Algorithm 2), the optimization point x^* that proved a split was possible is used as the value of `simulation` in the new set. Also, when an affine transformation of the set is computed (line 12 in Algorithm 2), or when the set is projected to zero, `simulation` must also be modified by the same transformation.

With a concrete value of x_n available in `simulation`, we use its sign to decide whether to first check the upper or lower bound of dimension n in θ. If the nth element of `simulation` is positive, for example, we first compute the lower bound. If this is positive (or zero), then get_sign can return pos. If the lower bound is negative, then we can immediately return posneg without solving another LP, since the simulation serves as a witness that x_n can also be positive. Only when the simulation value of x_n is zero do we need to solve two LPs.

We call this method Sim in Fig. 2. This is shown to be generally faster than the previous methods, as the overhead to track simulations is small compared with the gains of solving fewer LPs.

Correctness Justification: If the lower bound of x_n is greater than zero, than its upper bound will be also be greater than zero and `pos` is the correct output. If the lower bound is less than zero and the nth element of `simulation` is greater than zero, than the upper bound will also be positive, since it must be greater than or equal to the value in the simulation (`simulation` is always a point in the set θ), and so `posneg` is correct. Similar for the opposite case.

3.4 Zonotope Prefilter

We can further reduce LP solving by using a zonotope. In each computation-state tuple s, we add a zonotope field z that overapproximates θ, so that $\theta \subseteq z$. In the ACAS Xu benchmarks (and most current benchmarks for verification of NNs), the input set of states is provided as interval values on each input, which is a box and can be used to initialize the zonotope. Otherwise, LPs can be solved to compute box bounds on the input set to serve as an initial value. During the affine transformation of θ (line 12 in Algorithm 2), the zonotope also gets the same transformation applied. Cases where θ gets projected to zero are also affine transformations and can be exactly computed with the zonotope z. The only unsupported operation in the algorithm for zonotopes is `add_constraint`, used during the splitting operation (lines 28–29 in Algorithm 2). We skip these operations for the zonotope, which is why z is an overapproximation of θ.

With a zonotope overapproximation z available during `get_sign`, we can sometimes reduce the number of LPs to zero. Computing the minimum and maximum of the n-th dimension of z is an optimization problem over zonotopes, which recall from Sect. 2.3 can be done efficiently as a simple summation. If the n-th dimension of z is completely positive or negative, we can return `pos` or `neg` immediately. Otherwise, if both positive and negative values are possible in the zonotope, we fall back to LP solving on θ to compute the possible signs. This can be done either by computing both bounds, which we call Zono-Bounds or with the simulation optimization from before, which we call Zono-Sim. The performance of the methods are shown in Fig. 2. The Zonotope-Sim method performs the fastest, verifying about 145 benchmarks in under 10 min and demonstrating that reduction in LP solving is worth the extra bookkeeping.

Correctness Justification: Rejecting branches without LP solving is justified by the fact that z is an overapproximation of θ. This is initially true, as if the input set is a box then $z = \theta$ and otherwise z is the box overapproximation of θ. This is also true for every operation other than `add_constraint`, as these are exact for zonotopes. Finally, it is also true when `add_constraint` operation is skipped on z, as adding constraints can only reduce the size of the set θ. If $\theta \subseteq z$, every smaller set θ' will also be a subset of z by transitivity, $\theta' \subseteq \theta \subseteq z$, and so an overapproximation is maintained by ignoring these operations with z. Finally, if the n-th dimension of an overapproximation of θ is strictly positive (or negative), the n-th dimension of θ will also be strictly positive (or negative).

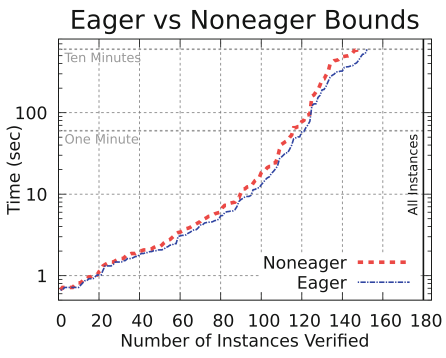


Fig. 3. Computing neuron output bounds eagerly improves speed.

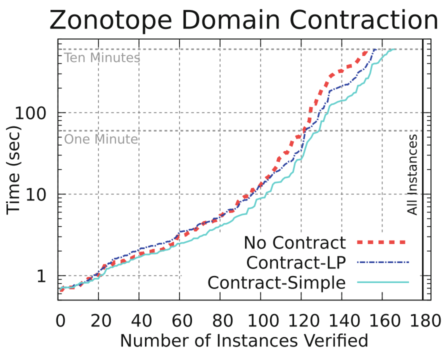


Fig. 4. Zonotope domain contraction improves overall performance.

3.5 Eager Bounds Computation

The `step` function shown in Algorithm 2 computes the sign of x_n for the current neuron n. An alternative approach is to compute the possible signs for every neuron's output in the current layer immediately after the affine transformation on line 12. These bounds can be saved in the computation-state tuple s and then accessed by `get_sign`. The potential advantage is that, if a split is determined as impossible for some neuron n, and a split occurs at some earlier neuron $i < n$, then the split will also be impossible for neuron n in both of the sets resulting from the earlier split at neuron i. In this way, computing the bounds once for neuron n is sufficient in the parent set, as opposed to computing the bounds twice, in each of the two children sets resulting from the split. The benefit can be even more drastic if there are multiple splits before neuron n is processed, where potentially an exponential number of bounds computations can be skipped due to a single computation in the parent. On the other hand, if a split is possible, we will have computed more bounds than we needed, as we will do the computation once in the parent and then once again in each of the children. Furthermore, this method incurs additional storage overhead for the bounds, as well as copy-time overhead when computation-state tuples are deep copied on line 27. Experiments are important to check if the benefits outweigh the costs.

The modified algorithm, which we call Eager, will use the zonotope prefilter and simulation as before to compute the bounds, but this will be done immediately after the affine transformation on line 12. Further, when a split occurs along neuron n in the posneg case, the bounds also get recomputed in the two children for the remaining neurons in the layer, starting at the next neuron $n+1$. Neurons where a split was already rejected do not have their bounds recomputed. This algorithm is compared with the previous approach, called Noneager. In Fig. 3, we see eager computation of bounds slightly improves performance.

Correctness Justification: When sets are split in the posneg case in Algorithm 2, each child's θ is a subset of the parent's θ. Thus, the upper and lower bound of the output of each neuron n can only move inward. Thus, if the parent's bounds for some neuron are strictly positive (or negative), then the two childrens' bounds will match the parent's and do not need to be recomputed.

3.6 Zonotope Contraction

The accuracy of the zonotope prefilters is important, as large overapproximation error will lead to the computed overapproximation range of x_n in zonotope z always overlapping zero, and thus performance similar to the Sim method. This effect is observed near the top of the curves in Fig. 2.

In order to improve accuracy, we propose a zonotope domain contraction approach, where the size of the zonotope set z is reduced while still maintaining an overapproximation of the exact star set θ. As discussed before, computing exact intersections of zonotopes is generally impossible when splitting (lines 28–29 in Algorithm 2). However, we can lower our expectations and instead consider other ways to reduce the size of zonotope z while maintaining $\theta \subseteq z$.

To do this, we use a slightly different definition of a zonotope, which we refer to as an *offset zonotope*. Instead of an affine transformation of the $[-1, 1]^p$ box, an offset zonotope is an affine transformation of an arbitrary box, $[l_1, u_1] \times \ldots \times [l_p, u_p]$, where each upper bound u_i is greater than or equal to the lower bound l_i. As this corresponds to an affine transformation of the $[-1, 1]^p$ box, offset zonotopes are equally expressive as ordinary zonotopes. Optimization over offset zonotopes can also be done using a simple summation, but instead of using Eq. 2, we use the following modified equation:

$$x_i^* = \begin{cases} u_i v_i, & \text{if } u_i v_i \cdot g_i \geq l_i v_i \cdot g_i \\ l_i v_i, & \text{otherwise} \end{cases} \tag{5}$$

Using offset zonotopes allows for some memory savings in the algorithm. The initial zonotope can be created using a zero vector as the zonotope center and the identity matrix as the generator matrix, the same as the initial input star set. In fact, with this approach, since the affine transformations being applied to the zonotope z and star set θ are identical, the centers and generator matrices will always remain the same, so that we only need to store one copy of these.

Beyond memory savings, with offset zonotopes we can consider ways to reduce the zonotope's overapproximation error when adding constraints to θ. The proposed computations are done after splitting (lines 28–29 in Algorithm 2), each time an extra constraint gets added to the star set's polytope P. The new linear constraint in the output space ($x_n \leq 0$ or $x_n \geq 0$) is transformed to a linear constraint in the initial space using Eq. 4. We then try to contract the size of the zonotope's box domain by increasing each l_i and reducing each u_i, while still maintaining an overapproximation of the intersection. We consider two ways to do this which we call Contract-LP and Contract-Simple.

In Contract-LP, linear programming is used to adjust each l_i and u_i. Since the affine transformations for the star set θ and the zonotope z are the same, z is an overapproximation if and only if the star set's polytope P is a subset of z's initial

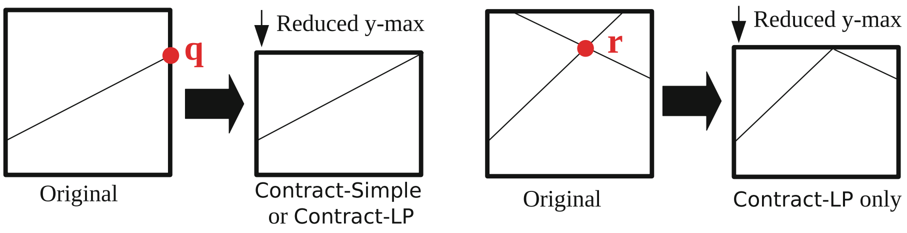


Fig. 5. Both Contract-Simple and Contract-LP can find point q to contract a zonotope's initial box (left), but only Contract-LP can find point r (right), as it requires reasoning with multiple linear constraints.

domain box $[l_1, u_1] \times \ldots \times [l_p, u_p]$. Thus, we can compute tight box bounds on P using linear programming, and using this box as the offset zonotope's initial domain box. This will be the smallest box that is possible for the current affine transformation while still maintaining an overapproximation. This approach, however, requires solving $2p$ linear programs, which may be expensive.

Another approach is possible without invoking LP, which we call Contract-Simple. Contract-Simple overapproximates the intersection by considering only the new linear constraint. This is a problem of finding the smallest box that contains the intersection of an initial box and a single halfspace, which can be solved geometrically without LP solving (see Appendix A for an algorithm).

Since Contract-Simple only considers a single constraint, it can be less accurate than Contract-LP. An illustration of the two methods is given in Fig. 5, where the initial domain is a two-dimensional box. The thin lines are the linear constraints that were added to θ, where all points below these lines are in the corresponding halfspaces. On the left, both Contract-Simple and Contract-LP can reduce the upper bound in the y direction by finding the point q, which lies at the intersection of one side of the original box domain and the new linear constraint. On the right, two constraints were added to the star θ (after two split operations), and they both must be considered at the same time to find point r to be able to reduce the upper bound in the y direction. In this case, only Contract-LP will succeed, as Contract-Simple works with only a single linear constraint at a time, and intersecting the original box with each of the constraints individually does not change its size.

Comparing the performance of the methods in Fig. 4, we see that the less-accurate but faster Contract-Simple works best for the ACAS Xu benchmarks. We expect both methods to take longer when the input set has more dimensions, but especially Contract-LP since it requires solving two LPs for every dimension.

Correctness Justification: The domain contraction procedures reduces the size of zonotope z while maintaining an overapproximation of the star set θ. This can be seen since the affine transformations in z and θ are always the same, and every point in the star set's initial input polytope P is also a point in the initial box domain of z. Since an overapproximation of θ is maintained, it is still sound to use z when determining the possible signs of a neuron's output.

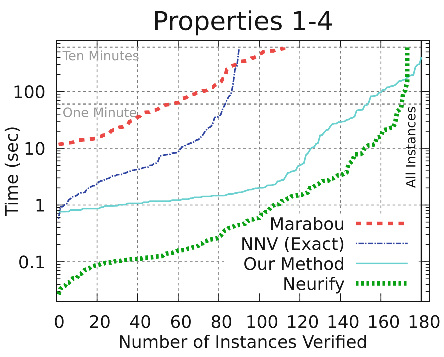


Fig. 6. Our method verifies all the benchmarks, although Neurify is usually faster when it completes.

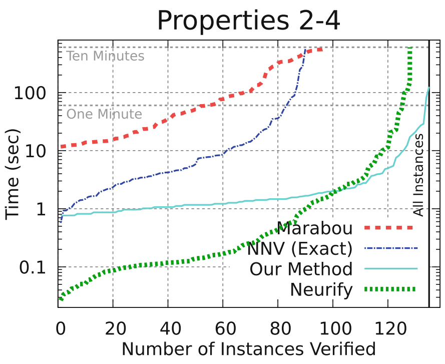


Fig. 7. Without property 1, our approach is generally fastest when the runtime exceeds two seconds.

4 Evaluation with Other Tools

We next compare the optimized implementation with other neural network verification tools. Our optimizations are part of the exact analysis mode of the NNENUM tool available at https://github.com/stanleybak/nnenum. The artifact evaluation package for our measurements here is online at http://stanleybak.com/papers/bak2020cav_repeatability.zip.

We evaluate with the fully optimized method, using DFS local search, Zono-Sim prefilter, Eager bounds, Contract-Simple zonotope domain contraction. Further, we use a parallelized version of the algorithm, where the details of the parallalization are provided in Appendix B. With a 12-thread implementation (one for each core on our evaluation system), the algorithm can now verify all 180 ACAS Xu benchmarks from properties 1–4 within the 10 min timeout. All measurements are done on our Laptop system, with hardware as described in the first paragraph of Sect. 3. The complete measurement data summarized here is available in Appendix D.

ACAS Xu Properties 1–4. We compare our method with Marabou [14] Neurify [26], and NNV [25]. Marabou is the newer, faster version of the Reluplex algorithm [13], where a Simplex-based LP solver is modified with special ReLU pivots[2]. Neurify is the newer, 20x faster version of the ReluVal algorithm [27], which does interval-based overapproximation, and splits intervals based on gradient information, ensuring the overapproximation error cannot cause to an incorrect result. NNV is the original Matlab implementation of the path enumeration method with star sets, available online at https://github.com/verivital/nnv. The verification result is consistent between the methods, which is a good sanity check for implementation correctness.

[2] For Marabou, we used the faster parallel divide-and-conquer mode with arguments as suggested in the paper [14]: `--dnc --initial-divides=4 --initial-timeout=5 --num-online-divides=4 --timeout-factor=1.5 --num-workers=12`.

Table 1. Tool runtime (secs) for ACAS Xu properties 5–10.

Property	Net	Result	Our method	ERAN	Neurify	NNV exact	Marabou
5	1-1	`SAFE`	13	–	12	671	1969
6.1	1-1	`SAFE`	67	–	3	6230	12425
6.2	1-1	`SAFE`	76	–	1	7612	17755
7	1-9	`UNSAFE`	5948	–	804	–	–
8	2-9	`UNSAFE`	.7	–	64	–	–
9	3-3	`SAFE`	88	318	393	12576	15235
10	4-5	`SAFE`	12	–	1	457	2795

The comparison on ACAS Xu benchmarks on properties 1–4 is shown in Fig. 6. Our method is the only approach able to analyze all 180 benchmarks in less than 10 min, and outperforms both Marabou and NNV.

The comparison with Neurify is more complicated. In Fig. 6, Neurify was faster (when it finished) on all but the largest instances. One advantage of Neurify compared with the other tools is that if the unsafe set is very far away from the possible outputs of a neural network, it can prove safety quickly with a very coarse overapproximation. Path enumeration methods, on the other hand, explore all paths regardless of the distance to the unsafe set. This is especially relevant for ACAS Xu property 1, where the system is unsafe if the first output, clear-of-conflict, is greater than 1500 whereas, for example on network 1-1, this output is always smaller than 1. The meaning of this property is also strange: the absolute value of a specific output is irrelevant, as relative values are used to select the current advisory. Neurify is admittedly the clear winner for all the networks with this property.

When this property is excluded and instead only the more difficult properties 2–4 are considered (Fig. 7), a different trend emerges. Here, our method outperforms Neurify when analysis takes more than about two seconds, which we believe is an encouraging result. Further, part of the reason why Neurify can be very quick on the easier benchmarks (with runtime less than two seconds) is that our implementation incurs a startup delay of about 0.6 s simply to start the Python process and begin executing our script, by which time the C++-based Neurify can verify 80 benchmarks. We believe the more interesting cases are when the runtimes are large, and we outperform Neurify in these cases.

Finally, we compare with using single-set overapproximations for analysis. NNV provides an approximate-star method, where rather splitting, a single star set is used to overapproximate the result of ReLU operations. While fast when it succeeds, this strategy can only verify 68 of the 180 benchmarks. Furthermore, the benchmarks it verified were also quickly checked with exact path enumeration. Of the 68 verified benchmarks, the largest performance difference was property 3 with network 3-3, which took 3.1 s with exact enumeration and 1.2 s with single-set overapproximation. For these ACAS Xu benchmarks, overapproximation using a single set does not provide much benefit.

Other ACAS Xu Properties. Another recently proposed and well received analysis method is presented in the elegant framework of abstract interpretation using zonotopes, in tools such as AI^2 [7] or DeepZ [21]. These methods are single-set overapproximation methods, similar to the approximate-star method in NNV, but with strictly more error (see Fig. 2 in the NNV paper [24] and the associated discussion). As these methods have more error than approximate-star, and since approximate-star could only verify 68 of the 180 benchmarks, we do not expect these methods to work well on the ACAS Xu system.

However, a recent extension to these methods has been proposed where the overapproximation is augmented with MILP solving [22] to provide complete analysis. This has been implemented in the ERAN tool, publicly available at https://github.com/eth-sri/eran. According to current version of the `README`, ERAN currently only supports property 9 of ACAS Xu, so we were unable to try this method on the other ACAS Xu networks or properties. Verifying property 9 uses a hard-coded custom strategy of first partitioning the input space into 6300 regions and analyzing these individually. This problem-specific parameter presents a problem for fair timing comparison, as the time needed to find the splitting parameter value of 6300 is unknown and does not get measured.

Ignoring this issue, we ran a comparison on property 9 and network 3-3, the only network where the property applies. A runtime comparison for ERAN[3] and the other tools is shown in Table 1. Surprisingly, our enumeration method significantly outperforms the overapproximation and refinement approaches both in Neurify and ERAN on this benchmark. Notice, however, that the original enumeration method in NNV is much slower than our method (about 150x slower in this case). Without the optimizations from this work, one would reach the opposite conclusion about which type of method works better for this benchmark. Both NNV and our method, however, report exploring the same number of paths, 338600 on this system.

For completeness, Table 1 also includes the other original ACAS Xu properties, which were each defined over a single network[4]. Both our method and Neurify completed all the benchmarks, although neither was best in all cases. Property 7 is particularly interesting, since the input set is the entire input space, so the number of path is very large. Hundreds of millions of paths were explored before finding a case where the property was violated.

5 Related Work

As the interest in neural networks has surged, so has research in their verification. We review some notable results here, although recent surveys may provide more

[3] For ACAS Xu analysis, we used the following arguments provided by the ERAN authors: `--domain deepzono --dataset acasxu --complete True`.

[4] Property 6's input set was a disjunction of two boxes which we split into two cases.

a thorough overview [15,28]. Verification approaches for NNs can broadly be characterized into geometric techniques, SMT methods, and MILP approaches.

Geometric approaches, like this work, propagate sets of states layer by layer. This can be done with polytopes [6,29] using libraries like the multi-parametric toolbox (MPT) [10], although certain operations do not scale well, in particular, affine transformation. Other approaches use geometric methods to bound the range of a neural network. These include AI^2 [7] and DeepZ [21] which propagate zonotopes through networks and are presented in the framework of abstract interpretation. ReluVal [27] and Neurify [26] also fall into this category, using interval symbolic methods to create overapproximations, followed by a refinement strategy based on symbolic gradient information. Some of these implementations are also sound with respect to floating-point rounding errors, which we have not considered here, mostly for lack of an LP solver that is both fast and does outward rounding. Other NN verification tools such as Reluplex, Marabou, ERAN, and NNV also use numeric LP solving. Another performance difference is that we used the free GLPK library for LP solving and some other tools used the commercial Gurobi optimizer, which is likely faster. Other refinement approaches partition the input space to detect adversarial examples [11], compute maximum sensitivity for verification [30], or perform refinement based on optimization shadow prices [20].

Mixed integer-linear programming (MILP) solvers can be used to exactly encode the reachable set of states through a ReLU network using the big-M trick to encode the possible branches [16,23]. This introduces a new boolean variables for each neuron, which may limit scalability. The MILP approach has also been combined with a local search [5] that uses gradient information to speed up the search process.

SMT approaches include the Reluplex [13] and Marabou [14], which modify the Simplex linear programming algorithm by splitting nodes into two, which are linked by the semantics of a ReLU. The search process is modified with updates that fix the ReLU semantics for the node pairs. Another tool, Planet, combines the MILP approach with SAT solving and linear overapproximation [6].

Here, we focused on input/output properties of the neural network, given as linear constraints. This formulation can check for adversarial examples [9] in image classification within some L_∞ norm of a base image, which are essentially box input sets. Other more meaningful semantic image perturbations such as rotations, color shifting, and lighting adjustments can also be converted into input/output set verification problems [19].

6 Conclusions

One of the major successes of formal verification is the development of fast model checking algorithms. When talking about how improvements to model checking algorithms came about, Ken McMillan noted:

> "Engineering matters: you can't properly evaluate a technique without an efficient implementation." [18]

With this in mind, we have strived to improve the practical efficiency of the complete path-enumeration method for neural network verification. Although the geometric path-enumeration method has been proposed before, we have shown that, by a sequence of optimizations, the method's scalability can be improved by orders of magnitude.

One limitation is that we have focused on the ACAS Xu benchmarks. Although there is a risk of overfitting our optimizations to the benchmarks being considered, we believe these benchmarks are fairly general in that they contain a mix of safe and unsafe instances, where the original verification times varied from seconds to days. In particular, we believe these networks are similar to others being used in control tasks, in terms of number of inputs and network size. Further, practical considerations prevent us from considering too many more benchmarks; our measurements already need over five days to run.

Unreported here, we were also able to run the implementation on larger perception networks to analyze L_∞ perturbation properties, networks with thousands of neurons and hundreds of inputs, which succeeds when the perturbation is sufficiently small. However, we believe path enumeration is the wrong approach for those systems, as the number of paths quickly becomes too large to enumerate. Instead, overapproximation and refinement methods would likely work best, and evaluating optimizations for these methods may be done in future work. One interpretation of the results presented here is that overapproximation and refinement methods still have significant room for improvement, as it is sometimes faster to explicitly enumerate benchmarks with millions of paths.

Many of the tools we have compared against also support more complicated network structures, with different layer types and nonlinear activation functions, whereas we only focused on the subclass of networks with ReLUs and fully-connected layers. We believe that this is an important enough subclass of neural networks that the results are still meaningful. Once the neural network verification community is more mature, we expect a standard input format and a set of categorized benchmarks will arise, similar to what has happened in the SMT [2], software verification [3], and hybrid systems [1] communities.

Acknowledgment. The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers SHF 1910017 and FMitF 1918450, the Air Force Office of Scientific Research (AFOSR) through contract numbers FA9550-18-1-0122 and FA9550-19-1-0288, the Air Force Research Laboratory (AFRL) under prime contract FA8650-15-D-2516 and the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFOSR, DARPA, or NSF. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

A Box Bounds Algorithm for Box-Halfspace Intersection

The problem of computing the box bounds of an intersection of an initial box and a single halfspace can be computed without LP. Consider a p-dimensional initial box defined with lower and upper bounds $[l_1, u_1] \times \ldots \times [l_p, u_p]$. Call the constraint defining the halfspace $f\alpha \leq \mathsf{g}$, where α is a p-dimensional vector of variables, f is a p-dimensional vector with entries $f_1, \ldots, f_p$, and g is a scalar.

Based on the signs of the signs of $f_1, \ldots, f_p$, we first find the vertex v^* in the box that minimizes the dot product $f \cdot v^*$. This can be done by choosing the ith element of v^* as:

$$v_i^* = \begin{cases} l_i, & \text{if } f_i \geq 0 \\ u_i, & \text{otherwise} \end{cases} \tag{6}$$

If $f \cdot v^* > \mathsf{g}$, then the intersection is the empty set. Otherwise, we attempt to contract in each of the p dimensions one-by-one.

For dimension i, if the lower bound was used to define v_i^*, then we attempt to decrease u_i. If the upper bound was used to define v_i^*, then we attempt to increase l_i. This is done by finding the point on the edge of the box which intersects the halfspace (point q in Fig. 5). Without loss of generality, assume the lower bound of dimension i defined v_i^*. The intersection point q is given by $(v_1^*, v_2^*, \ldots x, \ldots v_p^*)$, where value of the ith coordinate, x, can be determined from the single-variable equation $q \cdot f = \mathsf{g}$. If f_i was zero, then this equation has no solution, and we cannot contract in this dimension (the half-space and the box edge where q must lie do not intersect). Otherwise, if we solve for x and find $x < u_i$, then we reduce u_i, setting it to x. The process repeats for every other dimension.

B Parallelization

The proposed approach can be parallelized in many ways. Here, we propose and evaluate a work-stealing strategy, where each thread maintains a local set of computation-state tuples and runs the high-level algorithm. Periodically, the number of tuples in each local set are communicated using a shared data structure, and if some worker thread has no work remaining, the other threads will push some of their local computation-state tuples to a shared global queue.

For this evaluation, we used the usual system setup described in the first paragraph of Sect. 3, which we label Laptop. In addition, to see the effect of more cores, we rented a `c5.metal` EC2 instance from Amazon Web Services, which we refer to as AWS Server. This setup ran Ubuntu 18.08, and included a dual Intel(R) Xeon(R) Platinum 8275CL processor running at 3.0 GHz, with a total of 48 physical cores (96 with hyperthreading) and 384 GB of main memory.

To evaluate parallelism, we needed to use a benchmark with sufficient difficulty where computation time dominates. For this, we chose ACAS Xu network

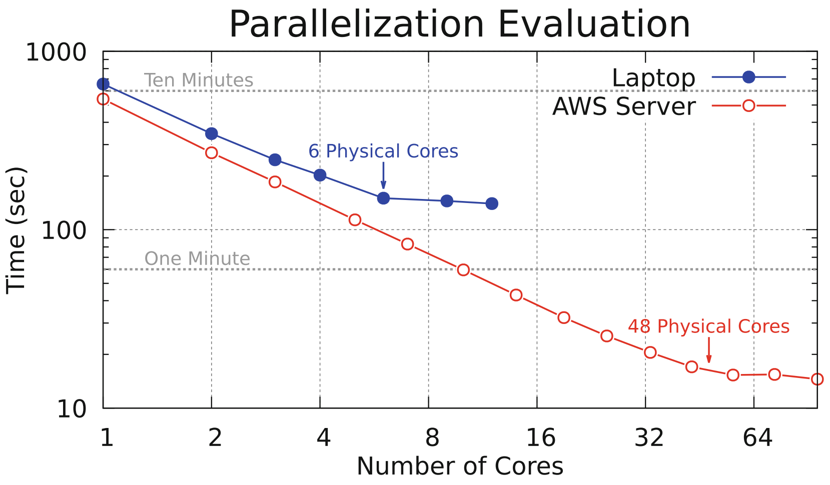


Fig. 8. Doubling the number of cores roughly halves the computation time, up to the physical core count on each platform.

4-2 with specification 2. In an earlier ACAS Xu evaluation [14], this property timed out (>55 min) or ran out of memory for every tool analyzed. The single-threaded runtime on the Laptop platform with our enumeration approach was 655 s (about 11 min), which enumerated 484555 paths in the network.

An evaluation where we adjusted the number of cores available to the computation process for each of the two platforms is shown in Fig. 8. The AWS Server platform was faster than the Laptop setup and, with all the cores being used, could enumerate the same 484555 paths in about 15 s. The linear trend on the log-log graph shows continuous improvement as more cores are added, up to the physical-core limit on each platform. The gains from hyperthreading are comparatively smaller. Even using all the cores, about 90% of the computation time was in the `step` function, as opposed to managing shared state. With more cores, further improvement through additional parallelization is likely possible.

Correctness Justification: Parallelization explores the same set of states, just in a different order.

C Full Optimization Data

See Table 2.

Table 2. Runtimes (sec) for each optimization. Dashes (—) are timeouts (10 min).

Prop	Net	Result	BFS	Copy	Bound	Sim	Zono-B	Zono-S	Eager	Con-LP	Con-Sim	Par
1	1-1	SAFE	—	—	399	166	359	159	129	65	50	10
1	1-2	SAFE	—	—	467	206	416	191	154	76	57	12
1	1-3	SAFE	—	—	—	485	—	496	375	197	163	32
1	1-4	SAFE	—	—	—	558	—	538	407	271	177	36
1	1-5	SAFE	—	—	—	492	—	491	360	215	138	30
1	1-6	SAFE	—	—	—	—	—	—	—	—	445	95
1	1-7	SAFE	—	—	539	250	518	259	190	113	78	17
1	1-8	SAFE	—	—	—	409	—	434	287	188	128	27
1	1-9	SAFE	—	—	—	476	—	446	324	221	132	29
1	2-1	SAFE	—	—	—	—	—	—	523	343	216	47
1	2-2	SAFE	—	—	—	—	—	—	—	—	599	119
1	2-3	SAFE	—	—	—	—	—	—	564	332	227	47
1	2-4	SAFE	—	—	—	383	—	412	272	193	120	27
1	2-5	SAFE	—	—	—	—	—	—	—	—	—	188
1	2-6	SAFE	—	—	—	—	—	—	—	517	400	82
1	2-7	SAFE	—	—	—	—	—	—	—	—	—	195
1	2-8	SAFE	—	—	—	—	—	—	—	—	—	163
1	2-9	SAFE	—	—	—	—	—	—	—	—	—	271
1	3-1	SAFE	—	—	—	—	—	—	438	411	263	57
1	3-2	SAFE	—	—	—	—	—	—	521	308	214	46
1	3-3	SAFE	—	—	—	—	—	—	—	596	390	84
1	3-4	SAFE	—	—	—	442	—	438	323	221	141	30
1	3-5	SAFE	—	—	—	—	—	—	—	—	401	86
1	3-6	SAFE	—	—	—	—	—	—	—	—	—	297
1	3-7	SAFE	—	—	—	—	—	—	—	—	—	155
1	3-8	SAFE	—	—	—	—	—	—	—	—	—	141
1	3-9	SAFE	—	—	—	—	—	—	—	—	507	107
1	4-1	SAFE	—	—	—	—	—	—	—	—	517	107
1	4-2	SAFE	—	—	—	—	—	—	—	—	568	124
1	4-3	SAFE	—	—	—	537	—	508	396	233	160	34
1	4-4	SAFE	—	—	—	523	—	584	365	245	155	34
1	4-5	SAFE	—	—	—	—	—	—	—	—	573	119
1	4-6	SAFE	—	—	—	—	—	—	—	—	—	408
1	4-7	SAFE	—	—	—	—	—	—	—	—	—	195
1	4-8	SAFE	—	—	—	—	—	—	—	—	—	131
1	4-9	SAFE	—	—	—	—	—	—	—	—	—	304
1	5-1	SAFE	—	—	—	—	—	—	482	322	232	48
1	5-2	SAFE	—	—	—	—	—	—	—	426	303	64
1	5-3	SAFE	—	—	—	508	—	498	366	214	143	32
1	5-4	SAFE	—	—	—	305	—	289	211	136	98	21
1	5-5	SAFE	—	—	—	—	—	—	—	368	264	57
1	5-6	SAFE	—	—	—	—	—	—	—	—	—	176
1	5-7	SAFE	—	—	—	—	—	—	—	—	474	97
1	5-8	SAFE	—	—	—	—	—	—	—	—	—	153
1	5-9	SAFE	—	—	—	—	—	—	—	—	—	161
2	1-1	SAFE	—	—	404	159	368	165	128	67	46	10
2	1-2	UNSAFE	—	58	24	11	23	12	9	5	4	1
2	1-3	UNSAFE	—	463	192	74	177	78	58	32	26	21
2	1-4	UNSAFE	—	31	15	6	13	6	5	4	3	1
2	1-5	UNSAFE	—	4	2	1	1	1	1	.8	.8	1
2	1-6	UNSAFE	—	—	—	517	—	579	373	260	175	19
2	1-7	SAFE	—	—	557	234	520	255	193	111	79	17
2	1-8	SAFE	—	—	—	403	—	399	297	184	126	27
2	1-9	SAFE	—	—	—	431	—	472	317	206	136	29
2	2-1	UNSAFE	—	92	39	18	37	18	13	7	5	.9
2	2-2	UNSAFE	—	.7	.7	.7	.7	.7	.7	.7	.7	.8
2	2-3	UNSAFE	—	8	4	2	4	2	2	1	1	1
2	2-4	UNSAFE	—	4	2	1	2	1	1	1	.9	.9
2	2-5	UNSAFE	—	37	17	8	18	8	6	3	3	1
2	2-6	UNSAFE	—	284	146	58	144	65	48	25	18	8

(*continued*)

Table 2. (*continued*)

Prop	Net	Result	BFS	Copy	Bound	Sim	Zono-B	Zono-S	Eager	Con-LP	Con-Sim	Par
2	2-7	UNSAFE	—	506	250	85	256	96	78	43	30	.9
2	2-8	UNSAFE	—	51	26	10	24	10	9	5	4	2
2	2-9	UNSAFE	—	—	—	291	—	320	242	132	94	2
2	3-1	UNSAFE	—	190	68	31	50	24	20	12	9	4
2	3-2	UNSAFE	—	250	88	38	96	44	27	19	14	1
2	3-3	SAFE	—	—	—	—	—	—	—	590	409	83
2	3-4	UNSAFE	—	197	106	41	97	42	32	18	13	.9
2	3-5	UNSAFE	—	67	34	14	32	15	11	6	5	.9
2	3-6	UNSAFE	—	27	10	5	11	5	5	3	2	5
2	3-7	UNSAFE	—	49	25	11	25	12	9	5	4	1
2	3-8	UNSAFE	—	266	112	42	114	50	32	20	15	2
2	3-9	UNSAFE	—	20	11	5	10	5	4	2	2	2
2	4-1	UNSAFE	—	115	45	19	40	20	14	8	7	5
2	4-2	SAFE	—	—	—	—	—	—	—	—	597	125
2	4-3	UNSAFE	—	2	1	1	2	1	.9	.8	.8	.9
2	4-4	UNSAFE	—	39	17	7	19	8	6	4	3	2
2	4-5	UNSAFE	—	470	239	97	200	94	71	34	27	2
2	4-6	UNSAFE	—	139	64	25	71	28	22	11	9	2
2	4-7	UNSAFE	—	461	215	93	210	93	65	35	27	1
2	4-8	UNSAFE	—	322	162	60	163	67	49	22	16	.9
2	4-9	UNSAFE	—	—	390	164	413	180	121	73	56	5
2	5-1	UNSAFE	—	32	15	7	15	8	6	3	3	.9
2	5-2	UNSAFE	—	91	39	18	30	16	12	6	6	1
2	5-3	UNSAFE	—	—	—	460	—	487	316	201	141	24
2	5-4	UNSAFE	—	2	1	1	1	1	.9	.8	.8	.9
2	5-5	UNSAFE	—	261	107	48	111	46	36	19	14	2
2	5-6	UNSAFE	—	208	102	41	95	41	30	15	10	2
2	5-7	UNSAFE	—	107	52	21	53	22	18	8	7	2
2	5-8	UNSAFE	—	302	161	63	160	67	50	27	19	1
2	5-9	UNSAFE	—	—	477	189	472	218	163	81	61	1
3	1-1	SAFE	561	526	232	116	125	80	58	103	58	12
3	1-2	SAFE	534	533	233	116	104	65	50	64	43	9
3	1-3	SAFE	143	147	75	35	30	20	15	19	14	4
3	1-4	SAFE	77	73	40	19	8	6	5	7	5	2
3	1-5	SAFE	88	84	42	21	10	7	6	8	6	2
3	1-6	SAFE	21	22	12	6	3	3	2	3	2	1
3	1-7	UNSAFE	8	.7	.7	.7	.7	.7	.7	.7	.7	.8
3	1-8	UNSAFE	6	.7	.7	.7	.7	.7	.7	.7	.7	.8
3	1-9	UNSAFE	4	.7	.7	.7	.7	.7	.7	.7	.7	.8
3	2-1	SAFE	147	142	75	34	31	21	16	24	14	4
3	2-2	SAFE	59	55	30	14	12	8	6	10	6	2
3	2-3	SAFE	108	101	50	25	19	12	9	14	9	3
3	2-4	SAFE	6	6	4	2	1	1	1	1	1	1
3	2-5	SAFE	33	33	18	9	4	4	3	4	3	1
3	2-6	SAFE	5	5	4	2	1	1	1	1	.9	1
3	2-7	SAFE	17	16	11	5	3	2	2	2	2	1
3	2-8	SAFE	6	6	5	2	1	1	1	1	1	1
3	2-9	SAFE	4	4	3	2	.9	.9	.8	1	.9	.9
3	3-1	SAFE	57	53	25	12	11	7	5	9	6	2
3	3-2	SAFE	578	537	226	117	93	53	40	59	36	8
3	3-3	SAFE	128	128	65	31	22	14	11	13	11	3
3	3-4	SAFE	27	26	16	7	5	4	3	4	2	1
3	3-5	SAFE	16	16	10	5	2	2	2	2	2	1
3	3-6	SAFE	31	33	20	10	5	4	3	3	3	1
3	3-7	SAFE	2	2	2	1	.8	.8	.7	.8	.8	.8
3	3-8	SAFE	12	12	8	4	2	2	1	2	1	1
3	3-9	SAFE	16	15	10	5	3	2	2	2	2	1
3	4-1	SAFE	18	18	11	5	5	3	2	4	3	1
3	4-2	SAFE	189	187	88	43	44	24	19	25	16	4
3	4-3	SAFE	282	283	136	63	64	35	29	32	24	5
3	4-4	SAFE	12	11	7	4	2	1	1	2	1	1
3	4-5	SAFE	4	4	3	2	1	1	.9	1	.9	1
3	4-6	SAFE	33	34	20	10	7	5	4	4	3	1
3	4-7	SAFE	15	15	11	5	2	2	2	2	2	1
3	4-8	SAFE	11	12	8	4	2	1	1	2	1	1
3	4-9	SAFE	12	11	8	4	2	2	2	2	1	1
3	5-1	SAFE	97	91	50	25	19	12	9	14	9	3
3	5-2	SAFE	18	19	11	6	5	3	2	4	2	1

(*continued*)

Table 2. (*continued*)

Prop	Net	Result	BFS	Copy	Bound	Sim	Zono-B	Zono-S	Eager	Con-LP	Con-Sim	Par
3	5-3	SAFE	22	23	12	6	5	3	3	4	3	1
3	5-4	SAFE	11	11	7	4	2	2	1	2	1	1
3	5-5	SAFE	15	14	10	5	2	2	2	2	2	1
3	5-6	SAFE	23	21	14	7	3	3	2	3	2	1
3	5-7	SAFE	2	2	2	1	.8	.8	.7	.8	.7	.8
3	5-8	SAFE	37	38	24	10	6	4	4	5	3	1
3	5-9	SAFE	2	2	2	1	.9	.8	.7	.8	.8	.8
4	1-1	SAFE	149	150	72	34	33	22	16	23	16	4
4	1-2	SAFE	135	130	52	27	21	15	12	16	11	3
4	1-3	SAFE	95	96	44	23	18	12	10	13	9	3
4	1-4	SAFE	12	11	7	4	2	2	2	2	2	1
4	1-5	SAFE	81	84	42	20	12	9	8	9	7	2
4	1-6	SAFE	41	37	20	11	7	5	4	6	4	2
4	1-7	UNSAFE	6	.7	.7	.7	.7	.7	.7	.7	.7	.8
4	1-8	UNSAFE	7	.8	.7	.7	.7	.7	.7	.7	.7	.8
4	1-9	UNSAFE	5	.7	.7	.7	.7	.7	.7	.7	.7	.8
4	2-1	SAFE	38	41	21	11	7	5	5	7	4	2
4	2-2	SAFE	50	51	27	13	8	6	5	6	4	2
4	2-3	SAFE	9	9	6	3	2	2	2	2	1	1
4	2-4	SAFE	8	9	5	3	2	2	1	2	1	1
4	2-5	SAFE	28	27	14	7	6	4	4	4	3	1
4	2-6	SAFE	15	15	9	5	3	2	2	2	2	1
4	2-7	SAFE	7	7	5	3	1	1	1	1	1	1
4	2-8	SAFE	40	43	25	11	5	4	3	4	3	1
4	2-9	SAFE	3	3	3	2	.9	.9	.9	.9	.9	.9
4	3-1	SAFE	56	52	27	13	7	6	5	6	5	2
4	3-2	SAFE	63	61	31	15	12	9	7	11	7	2
4	3-3	SAFE	10	9	6	3	2	2	2	2	2	1
4	3-4	SAFE	12	12	7	3	2	2	2	2	2	1
4	3-5	SAFE	38	40	22	10	8	6	4	5	4	2
4	3-6	SAFE	20	20	12	6	3	3	2	3	2	1
4	3-7	SAFE	17	17	11	5	3	2	2	2	2	1
4	3-8	SAFE	7	7	5	2	2	2	1	1	1	1
4	3-9	SAFE	51	48	29	13	7	5	5	5	4	2
4	4-1	SAFE	7	7	5	3	2	1	1	2	1	1
4	4-2	SAFE	14	14	8	5	3	2	2	2	2	1
4	4-3	SAFE	26	27	14	8	5	4	3	5	3	1
4	4-4	SAFE	20	20	11	6	3	2	2	2	2	1
4	4-5	SAFE	17	16	9	5	3	2	2	2	2	1
4	4-6	SAFE	30	30	15	7	5	3	3	4	3	1
4	4-7	SAFE	3	3	2	1	1	.9	.9	.9	.8	.8
4	4-8	SAFE	24	23	16	7	4	3	2	3	2	1
4	4-9	SAFE	43	40	24	12	5	4	4	4	4	2
4	5-1	SAFE	57	53	26	14	10	7	6	8	5	2
4	5-2	SAFE	38	34	17	9	7	4	4	5	4	2
4	5-3	SAFE	14	13	8	4	3	2	2	3	2	1
4	5-4	SAFE	13	13	8	4	2	2	2	2	2	1
4	5-5	SAFE	17	17	11	6	3	3	2	2	2	1
4	5-6	SAFE	10	10	6	3	2	2	2	2	1	1
4	5-7	SAFE	3	3	2	1	.9	.8	.8	.9	.8	.8
4	5-8	SAFE	8	8	6	3	2	1	1	1	1	1
4	5-9	SAFE	14	13	8	4	2	2	2	2	2	1

D Full Tool Comparison Data

This section contains the complete data measured in the optimization improvements from Sect. 3 (Table 3).

Table 3. Runtimes (sec) for each tool. Dashes (—) are timeouts (10 min).

Prop	Net	Result	Marabou	NNV Exact	Our Method	Neurify
1	1-1	SAFE	95	—	10	**.1**
1	1-2	SAFE	168	—	12	**.2**
1	1-3	SAFE	—	—	32	**1**
1	1-4	SAFE	—	—	36	**2**
1	1-5	SAFE	119	—	30	**.2**
1	1-6	SAFE	110	—	95	**.2**
1	1-7	SAFE	63	—	17	**.1**
1	1-8	SAFE	56	—	27	**.1**
1	1-9	SAFE	43	—	29	**.1**
1	2-1	SAFE	—	—	47	**.6**
1	2-2	SAFE	—	—	119	**1**
1	2-3	SAFE	—	—	47	**1**
1	2-4	SAFE	294	—	27	**.5**
1	2-5	SAFE	—	—	188	**4**
1	2-6	SAFE	—	—	82	**3**
1	2-7	SAFE	—	—	195	**11**
1	2-8	SAFE	—	—	163	**3**
1	2-9	SAFE	—	—	271	**8**
1	3-1	SAFE	—	—	57	**.4**
1	3-2	SAFE	—	—	46	**.7**
1	3-3	SAFE	521	—	84	**1**
1	3-4	SAFE	510	—	30	**.6**
1	3-5	SAFE	—	—	86	**2**
1	3-6	SAFE	—	—	297	**28**
1	3-7	SAFE	—	—	155	**12**
1	3-8	SAFE	—	—	141	**8**
1	3-9	SAFE	—	—	107	**11**
1	4-1	SAFE	—	—	107	**16**
1	4-2	SAFE	—	—	124	**3**
1	4-3	SAFE	—	—	34	**1**
1	4-4	SAFE	387	—	34	**.7**
1	4-5	SAFE	—	—	119	**3**
1	4-6	SAFE	—	—	408	**22**
1	4-7	SAFE	—	—	195	**23**
1	4-8	SAFE	—	—	131	**47**
1	4-9	SAFE	—	—	304	**21**
1	5-1	SAFE	353	—	48	**.4**
1	5-2	SAFE	522	—	64	**.7**
1	5-3	SAFE	128	—	32	**.2**
1	5-4	SAFE	574	—	21	**.4**
1	5-5	SAFE	—	—	57	**1**
1	5-6	SAFE	—	—	176	**15**
1	5-7	SAFE	—	—	97	**3**
1	5-8	SAFE	—	—	153	**16**
1	5-9	SAFE	—	—	161	**8**
2	1-1	SAFE	—	—	10	**.6**
2	1-2	UNSAFE	254	—	**1**	3
2	1-3	UNSAFE	—	—	21	**11**
2	1-4	UNSAFE	—	—	**1**	10
2	1-5	UNSAFE	—	—	**1**	—
2	1-6	UNSAFE	—	—	**19**	52
2	1-7	SAFE	—	—	17	**6**
2	1-8	SAFE	—	—	27	**23**
2	1-9	SAFE	—	—	29	**11**
2	2-1	UNSAFE	59	—	.9	**.1**

(*continued*)

Table 3. (*continued*)

Prop	Net	Result	Marabou	NNV Exact	Our Method	Neurify
2	2-2	UNSAFE	—	—	.8	**.1**
2	2-3	UNSAFE	549	—	1	**.1**
2	2-4	UNSAFE	18	—	.9	**.1**
2	2-5	UNSAFE	547	—	1	**.1**
2	2-6	UNSAFE	—	—	8	**.1**
2	2-7	UNSAFE	24	—	.9	**.1**
2	2-8	UNSAFE	102	—	2	**.1**
2	2-9	UNSAFE	—	—	**2**	—
2	3-1	UNSAFE	97	—	4	**.1**
2	3-2	UNSAFE	345	—	**1**	—
2	3-3	SAFE	—	—	**83**	—
2	3-4	UNSAFE	—	—	.9	**.1**
2	3-5	UNSAFE	319	—	.9	**.1**
2	3-6	UNSAFE	471	—	5	**.1**
2	3-7	UNSAFE	—	—	**1**	—
2	3-8	UNSAFE	—	—	2	**.1**
2	3-9	UNSAFE	457	—	2	**.1**
2	4-1	UNSAFE	—	—	5	**.2**
2	4-2	SAFE	—	—	**125**	—
2	4-3	UNSAFE	566	—	.9	**.1**
2	4-4	UNSAFE	288	—	2	**.1**
2	4-5	UNSAFE	—	—	2	**.1**
2	4-6	UNSAFE	419	—	2	**.1**
2	4-7	UNSAFE	—	—	1	**.1**
2	4-8	UNSAFE	336	—	.9	**.1**
2	4-9	UNSAFE	—	—	**5**	45
2	5-1	UNSAFE	119	—	.9	**.1**
2	5-2	UNSAFE	24	—	1	**.1**
2	5-3	UNSAFE	—	—	**24**	—
2	5-4	UNSAFE	360	—	.9	**.1**
2	5-5	UNSAFE	278	—	2	**.1**
2	5-6	UNSAFE	547	—	2	**.1**
2	5-7	UNSAFE	17	—	2	**.1**
2	5-8	UNSAFE	246	—	1	**.1**
2	5-9	UNSAFE	47	—	1	**.1**
3	1-1	SAFE	—	564	**12**	104
3	1-2	SAFE	—	283	9	**2**
3	1-3	SAFE	—	58	4	**3**
3	1-4	SAFE	342	12	2	**.3**
3	1-5	SAFE	520	17	2	**.2**
3	1-6	SAFE	43	4	1	**.1**
3	1-7	UNSAFE	12	2	.8	**.1**
3	1-8	UNSAFE	12	2	.8	**.1**
3	1-9	UNSAFE	12	1	.8	**.05**
3	2-1	SAFE	—	70	**4**	21
3	2-2	SAFE	—	23	**2**	8
3	2-3	SAFE	—	39	**3**	3
3	2-4	SAFE	15	2	1	**.5**
3	2-5	SAFE	18	7	1	**.4**
3	2-6	SAFE	15	1	1	**.04**
3	2-7	SAFE	16	4	1	**.3**
3	2-8	SAFE	15	2	1	**.1**
3	2-9	SAFE	13	1	.9	**.03**
3	3-1	SAFE	406	21	**2**	3
3	3-2	SAFE	—	247	8	**6**
3	3-3	SAFE	—	35	3	**.2**
3	3-4	SAFE	47	8	1	**.4**
3	3-5	SAFE	15	4	**1**	5
3	3-6	SAFE	390	7	**1**	151
3	3-7	SAFE	13	.9	.8	**.1**
3	3-8	SAFE	36	3	**1**	4
3	3-9	SAFE	45	4	**1**	3
3	4-1	SAFE	—	8	**1**	8
3	4-2	SAFE	—	88	**4**	97
3	4-3	SAFE	—	130	5	**2**
3	4-4	SAFE	14	2	1	**.1**
3	4-5	SAFE	14	1	1	**.1**
3	4-6	SAFE	102	11	1	**.2**

(*continued*)

Table 3. (*continued*)

Prop	Net	Result	Marabou	NNV Exact	Our Method	Neurify
3	4-7	SAFE	96	3	1	**.6**
3	4-8	SAFE	85	2	**1**	2
3	4-9	SAFE	33	3	1	**.1**
3	5-1	SAFE	—	35	**3**	21
3	5-2	SAFE	—	8	**1**	2
3	5-3	SAFE	146	8	1	**.2**
3	5-4	SAFE	17	3	1	**.2**
3	5-5	SAFE	24	4	1	**.5**
3	5-6	SAFE	88	5	1	**.9**
3	5-7	SAFE	14	.6	.8	**.04**
3	5-8	SAFE	43	9	1	**.1**
3	5-9	SAFE	14	.9	.8	**.1**
4	1-1	SAFE	—	82	4	**1**
4	1-2	SAFE	—	50	3	**1**
4	1-3	SAFE	—	36	3	**.4**
4	1-4	SAFE	105	3	1	**.2**
4	1-5	SAFE	504	24	2	**.4**
4	1-6	SAFE	89	12	2	**.2**
4	1-7	UNSAFE	12	2	.8	**.1**
4	1-8	UNSAFE	12	2	.8	**.1**
4	1-9	UNSAFE	12	2	.8	**.1**
4	2-1	SAFE	171	14	2	**.8**
4	2-2	SAFE	520	14	**2**	2
4	2-3	SAFE	77	3	1	**.8**
4	2-4	SAFE	23	3	1	**.2**
4	2-5	SAFE	61	11	1	**.4**
4	2-6	SAFE	90	5	1	**.3**
4	2-7	SAFE	14	2	1	**.1**
4	2-8	SAFE	43	8	1	**.1**
4	2-9	SAFE	13	1	.9	**.03**
4	3-1	SAFE	—	13	2	**1**
4	3-2	SAFE	134	27	2	**.4**
4	3-3	SAFE	21	4	1	**.1**
4	3-4	SAFE	20	4	1	**.2**
4	3-5	SAFE	59	15	2	**1**
4	3-6	SAFE	66	5	**1**	2
4	3-7	SAFE	16	4	1	**.3**
4	3-8	SAFE	29	3	1	**.3**
4	3-9	SAFE	63	12	2	**1**
4	4-1	SAFE	78	3	**1**	3
4	4-2	SAFE	60	5	**1**	2
4	4-3	SAFE	134	10	1	**1**
4	4-4	SAFE	41	5	**1**	1
4	4-5	SAFE	62	4	**1**	2
4	4-6	SAFE	14	8	1	**.04**
4	4-7	SAFE	21	1	.8	**.2**
4	4-8	SAFE	37	6	1	**.2**
4	4-9	SAFE	25	8	2	**.1**
4	5-1	SAFE	339	19	**2**	3
4	5-2	SAFE	51	12	2	**.5**
4	5-3	SAFE	52	5	1	**.2**
4	5-4	SAFE	31	4	1	**.2**
4	5-5	SAFE	49	5	1	**.6**
4	5-6	SAFE	76	3	1	**.3**
4	5-7	SAFE	14	1	.8	**.04**
4	5-8	SAFE	31	3	1	**.1**
4	5-9	SAFE	26	3	1	**.1**

References

1. Althoff, M., et al.: ARCH-COMP19 category report: continuous and hybrid systems with linear continuous dynamics. In: ARCH 2019, 6th International Workshop on Applied Verification of Continuous and Hybrid Systems, pp. 14–40 (2019)
2. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England), vol. 13, p. 14 (2010)
3. Beyer, D.: Competition on software verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_38
4. Duggirala, P.S., Viswanathan, M.: Parsimonious, simulation based verification of linear systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 477–494. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_26
5. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_9
6. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
7. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI^2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18. IEEE (2018)
8. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
9. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
10. Herceg, M., Kvasnica, M., Jones, C.N., Morari, M.: Multi-parametric toolbox 3.0. In: 2013 European Control Conference (ECC), pp. 502–510. IEEE (2013)
11. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
12. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J.: Policy compression for aircraft collision avoidance systems. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), pp. 1–10. IEEE (2016)
13. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
14. Katz, G., et al.: The Marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
15. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. arXiv preprint arXiv:1903.06758 (2019)
16. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. arXiv preprint arXiv:1706.07351 (2017)

17. Marston, M., Baca, G.: ACAS-Xu initial self-separation flight tests (2015). http://hdl.handle.net/2060/20150008347
18. McMillan, K.: A perspective on formal verification. In: David Dill @ 60 Workshop, colocated with CAV (2017)
19. Mohapatra, J., Chen, P.-Y., Liu, S., Daniel, L., et al.: Towards verifying robustness of neural networks against semantic perturbations. arXiv preprint arXiv:1912.09533 (2019)
20. Royo, V.R., Calandra, R., Stipanovic, D.M., Tomlin, C.: Fast neural network verification via shadow prices. arXiv preprint arXiv:1902.07247 (2019)
21. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems, pp. 10802–10813 (2018)
22. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (ICLR 2019) (2019)
23. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. arXiv preprint arXiv:1711.07356 (2017)
24. Tran, H.-D., et al.: Star-based reachability analysis of deep neural networks. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 670–686. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_39
25. Tran, H.-D., et al.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 3–17. Springer, Cham (2020)
26. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems, pp. 6367–6377 (2018)
27. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium, pp. 1599–1614 (2018)
28. Xiang, W., et al.: Verification for machine learning, autonomy, and neural networks survey. arXiv preprint arXiv:1810.01989 (2018)
29. Xiang, W., Tran, H.-D., Johnson, T.T.: Reachable set computation and safety verification for neural networks with ReLU activations. arXiv preprint arXiv:1712.08163 (2017)
30. Xiang, W., Tran, H.-D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)

Systematic Generation of Diverse Benchmarks for DNN Verification

Dong Xu(✉), David Shriver, Matthew B. Dwyer, and Sebastian Elbaum

University of Virginia,
Charlottesville, VA 22904, USA
{dx3yy,dls2fc,matthewbdwyer,
selbaum}@virginia.edu

Abstract. The field of verification has advanced due to the interplay of theoretical development and empirical evaluation. Benchmarks play an important role in this by supporting the assessment of the state-of-the-art and comparison of alternative verification approaches. Recent years have witnessed significant developments in the verification of deep neural networks, but diverse benchmarks representing the range of verification problems in this domain do not yet exist. This paper describes a neural network verification benchmark generator, GDVB, that systematically varies aspects of problems in the benchmark that influence verifier performance. Through a series of studies, we illustrate how GDVB can assist in advancing the sub-field of neural network verification by more efficiently providing richer and less biased sets of verification problems.

Keywords: Neural network · Verification · Benchmark · Covering array

1 Motivation

Advances in machine learning have enabled training of deep neural networks (DNN) that are capable of realizing complex functions that rival or exceed the performance of human-built software, e.g., [27,32,41]. This success has led system developers to deploy, or consider deployment of, DNN models in critical systems, e.g., [12,39,53]. Consequently, the verification of correctness properties of DNNs has become a key challenge to assuring autonomous systems, and the research community has risen to this challenge. In the three years since Katz et al. [30] presented ReLuplex at CAV 2017, researchers have published more than 20 DNN verification approaches supporting different properties and DNN architectures and spanning a range of algorithmic approaches [9,13,14,18–20,22,29–31,36,45,46,50,56,59–63]. While DNN verification has its own unique challenges, it is also a recent example in the long-history of domain-specific verification research, e.g., for hardware [25], software [17], real-time systems [58], and cryptographic protocols [40], and can benefit from the experience of these communities.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 97–121, 2020.
https://doi.org/10.1007/978-3-030-53288-8_5

A key lesson learned by the community is that despite the fact that verification emphasizes the development of theoretical and algorithmic techniques, *advances in verification research often arise from understanding how different algorithmic and implementation approaches compare* – a process that requires empirical study. Empirical study in verification is common, but unlike many other fields of computer science, for decades it has organized *verification tool competitions* that serve as a regular and long-running form of community-driven empirical study. Researchers tracked the progress of SMT solvers over a span of 6 years at these community-driven empirical studies and found that repeatedly "a certain solver presents a key idea that improves the performance in a particular division, and this idea is implemented by most solvers" in the following year [7]. Enabling the type of comparative studies that drive such advances requires *verification benchmarks* – a fact that the verification community has recognized for at least 25 years, e.g., [8,10,33,43,55].

Benchmarking in verification has evolved in response to the demands of empirical study within the field, e.g., [1–4], to support two objectives: (A1) *assessment of the state-of-the-art* and (A2) *comparison of alternative approaches.* In support of these, the verification community has favored benchmarks that: (R1) **are diverse in structure and difficulty**; (R2) **represent verifier use cases**; and (R3) **evolve as verification technology advances**.

The verification benchmarking and competition literature suggests that these requirements are widely accepted. For example, the TPTP benchmark's stated goals include R1 ("contains problems varying in difficulty"), R2 ("spans a diversity of subject matters"), and R3 ("is up-to-date", "provides a mechanism for adding new problems") [54]. Moreover, these requirements are promoted, either explicitly or implicitly, by many of the regularly held verification competitions. To meet R1 and R2 SAT competitions construct benchmarks that include problems from six different domains: software, hardware, A.I, obstruction, combinatorial challenges, and theorem proving [4]. SAT competitions since 2017 have instituted a *bring your own benchmarks* policy that requires verifier developers to submit 20 new benchmarks with at least 10 that are "not too easy" or "too hard" – which helps to address R1 and R3. SMT competitions have used selection criteria that are biased towards these same requirements, e.g., "balancing the difficulty of benchmarks" [7].

Verification competitions have undoubtedly been a positive force for developing high-quality verification benchmarks, but prior to their existence researchers were forced to develop their own "benchmarks" – a collection of verification problems on which they evaluate their techniques and perhaps others. This is the situation that the subfield of *DNN verification* finds itself in.

The risk in letting technique developers choose their own benchmark is selection bias – that the selected problems do not represent a broad or important population of problems. For example, if an SMT benchmark were selected based on the constraints generated by symbolic execution tools they would be structurally biased, consisting only of conjunctive formula. As another example, if a SAT benchmark were generated randomly it is likely that a large portion of the benchmark would not represent realistic use cases.

Good benchmarks are expensive to develop, e.g., [11], but they are an invaluable resource for advancing a research community. When well designed they seek to balance requirements R1-R3 and to support a fair and accurate assessment of the state-of-the-art and comparison between alternative algorithmic and implementation approaches. This paper reports on GDVB, the first *framework for systematic **G**eneration of **D**NN **V**erification problem **B**enchmarks*, that meets the de-facto requirements for verification benchmarks, R1–R3, in order to support objectives A1–A2 for the rapidly evolving field of DNN verification.

GDVB takes a **generative** approach to benchmark development – an approach that has risen in popularity in recent years [5,35,64]. Unlike, other generative benchmark approaches GDVB seeks to systematically cover variations in verification problems that are known to influence verifier performance. Towards that end, GDVB is parameterized by: (1) a set of *factors* known to influence the performance of DNN verifiers; (2) a *coverage* goal that determines the combination of factors that should be reflected in the benchmark; and (3) a *seed* verification problem from which a set of variant problems are generated. From these parameters, it computes a constrained mixed-level covering array [15] defining a set of factor-value tuples. Each tuple defines how the seed verification problem can be transformed to give rise to a verification problem capable of exposing performance variation in a DNN verifier.

As a benchmark generator GDVB naturally meets requirement R3. By starting from a seed network representing a DNN verification use case, GDVB is guaranteed to meet R2. As we discuss in Sect. 4, the use of factors allows GDVB to produce systematically diverse verification problems both in terms of structure and difficulty in order to meet requirement R1. Moreover, GDVB offers the potential to reduce selection bias in performing evaluations of DNN verifiers, since it assures coverage of a space of performance related factors. Finally, GDVB is designed to support the rapidly evolving field of DNN verifiers by allowing the generation of benchmarks, e.g., from new seeds as verifiers improve, as new performance factors are identified, and to target challenge problems in different DNN domains, e.g., regression models for autonomous UAV navigation [39,53].

The contributions of this paper are: identification of the need for unbiased and diverse benchmarks for DNN verification; a study of factors that affect the performance of DNN verification tools (Sect. 3); the specification of a verification benchmark as the solution to a constrained mixed-level covering array problem (Sect. 4); the GDVB algorithm for computing a benchmark from a verification problem by transforming the neural network and correctness specification (Sect. 4.3); the evaluation of GDVB on multiple state-of-the-art DNN verifiers using different seed verification problems that demonstrates how GDVB results can support the evaluation of DNN verifiers (Sect. 5); and the GDVB tool.

2 Background and Related Wok

Deep Neural Networks (DNN). A DNN is trained to accurately approximate a target function, $f : \mathbb{R}^d \rightarrow \mathbb{R}^r$. A network, $n : \mathbb{R}^d \rightarrow \mathbb{R}^r$, is comprised of a graph of L hidden layers, $l_1, \ldots, l_L$, along with an input layer, $l_{in} = l_0$, and output layer, $l_{out} = l_{L+1}$. Each hidden layer defines an independent function, where their composition when applied to the output of l_{in} generates values in l_{out} that define the network output.

Hidden layers are, generally, comprised of a set of *neurons* that accumulate a weighted sum of their inputs from the prior layer and then apply an *activation function* to determine how to non-linearly scale that sum to compute the output from the layer. A variety of different activation functions have been explored in the literature, including: rectified linear units (ReLU), sigmoid, and tanh.

The design of a DNN involves choosing an appropriate set of *layer type*s, e.g., convolutional, maxpooling, fully-connected, the instantiation of those layers, e.g., the number of neurons, the specific activation function, and the definition of how layers are interconnected. Together these comprise the DNN *architecture* [23].

Networks are trained using a variety of algorithmic strategies with the goal of minimizing the loss in the approximation of the learned function relative to some proxy for f, e.g., labeled training data. The training process is stochastic, e.g., initial weight values are randomized, which leads to variation in n even when architecture, training algorithm, and training data are fixed.

Section 3 reveals how DNN architecture can influence verification performance.

DNN Specifications. Given a network $n : \mathbb{R}^d \rightarrow \mathbb{R}^r$, a property, ϕ, defines a set of constraints over the inputs, $\phi_{\boldsymbol{x}}$, and an associated set of constraints over the outputs, ϕ_y. Verification of n seeks to prove: $\forall \boldsymbol{x} \in \mathbb{R}^d : \phi_{\boldsymbol{x}}(\boldsymbol{x}) \Rightarrow \phi_y(\mathbf{N}(x))$ where $\mathbf{N}(x)$ is running the neural network n with input x.

Specifying behavioral properties of DNNs is challenging and is an active area of research [24]. In [30], a set of 188 purely conjunctive properties, of the form described above, were defined for a simple neural network, with 7 inputs, encoding of a rule set for autonomous aircraft collision avoidance (ACAS). In [44,59,60], properties expressing output range invariants were used, for example, that the steering angle never exceeded an absolute value of 30°. Much of the work on DNN verification has focused on local robustness properties [50–52], which state that for a selected target input the output of the network is invariant for other inputs within a specified distance of the target.

Section 3 reveals how the specification can influence verification performance.

DNN Verification Methods and Tools. There are a variety of different algorithmic and implementation approaches taken to verifying the validity of a DNN with respect to a stated correctness property.

Definition 1. *A DNN verification problem, $\langle n, \phi \rangle$, is comprised of a DNN, n, and a property specification, ϕ. The outcome of a verification problem for a DNN verifier indicates whether $n \models \phi$ is valid, invalid, or unknown – indicating that the problem cannot be determined to be either valid or invalid.*

A recent DNN verification survey [37], classifies approaches as being based on reachability, optimization, and search algorithms – or their combination. Reachability methods begin with a symbolic encoding of an input set and compute, for each layer, a symbolic encoding of the output set. They vary in the symbolic encodings used, e.g., intervals, polyhedra, and in the degree of overapproximation they introduce [22,46,50,63]. Optimization methods formulate verification as an optimization problem whose solution implies the validity of ϕ [9,19,38,45,56,62]. Search methods work in combination with reachability and optimization, by decomposing the input space to formulate verification sub-problems that are discharged by the above techniques [13,14,18,20,29,30,59–61].

In this paper, we use implementations of the following verifiers: ERAN [50], BaB [14], Neurify [59], Planet [20], and ReLuplex [30].

Verification Benchmarking. We covered the broad landscape of work on benchmark development for verification in (Sect. 1). There have been efforts to develop benchmarks within a variety of different verification problem domains, e.g. hardware [25], software [17], real-time systems [58], cryptographic protocols [40], and for different encodings of verification problems, e.g., model checking [33], SAT [4], SMT [8], and theorem proving [55].

In recent work on DNN verification, researchers have shared collections of examples that, in a sense, serve as informal benchmarks and permit comparative evaluation, e.g. [30,50]. While valuable, these examples were not intended to, and do not, comprise a benchmark meeting requirements R1–R3. To our knowledge, GDVB is the first approach to achieving those goals for DNN verification.

For several years, the SAT community has been exploring scalable benchmarks, e.g., [21,35]. For instance, to explore conflict-driven clause learning (CDCL) SAT solver performance, Elffers et al. [21] used crafted parameterized benchmarks that can be scaled with respect to different factors that may influence performance. We conduct a similar domain analysis of factors, but focus on the landscape of DNN verification algorithms developed to date. Like this line of work, GDVB advocates a scalable approach to benchmark generation. As described in Sect. 4, GDVB starts with seed problems that are challenging for current verifiers and "scales them down", but it can also be applied to start with easier seed problems and "scale them up" as more typical of the prior work on scalable benchmarking.

Verification Benchmark Ranking. The verification community has explored a variety of ranking schemes for assessing the cost-effectiveness of techniques. A key challenge is that verification techniques vary not only in their cost, e.g., time to produce a verification result, but also in their accuracy, e.g., whether they produce an *unknown* result. For example, SAT competitions have employed a range of scoring models, e.g., purse-based ranking, *solution-count ranking* (SCR), careful ranking, and penalized average runtime (PAR2) [6]. SCR, which counts the number of solved problem instances and uses verification time as a tie breaker [57], is the scoring system of choice [1,4]. In Sect. 5, we report DNN verifier performance using both SCR and PAR2 scoring systems.

Covering Arrays. In Sect. 3 we explore factors that influence DNN verifier performance. Studying all their combinations would be cost prohibitive, so we consider weaker notions of coverage.

A covering array defines a systematic method for testing how combinations of parameter values influence system performance [16]. A covering array is an $N \times k$ array. The k columns represent *factors* that may influence performance and cells can take on v *levels* – defining settings for factors. The N rows of the array define combinations of factor-levels. Arrays are defined to achieve a *strength* of the coverage, t. $t = 2$ defines pairwise strength, which means that all pairs of levels for all factors are present in some row of the covering array.

We require a richer form of covering array that permits the number of levels to vary with different factors, i.e., a mixed-level covering array (MCA), and that can constrain specified factor-level combinations, e.g., by forbidding their inclusion in the MCA. By modeling each factor as a variable and its levels as the domain of the variable, one can express constraints as propositional logic formulae over equality terms; if the levels are ordered then richer underlying theories can be applied. A constrained-MCA defines an MCA that is consistent with a given constraint, C.

Definition 2. *Constrained Mixed-level Covering Array (Definition 2.9 from [15])*
$CMCA(N; t, k, (|v_1|, |v_2|, ..., |v_k|), C)$ *is an* $N \times k$ *array on* $|v|$ *symbols, where* $|v| = \sum_{i=0}^{k} |v_i|$, *with the following properties: 1) Each column* $i (1 \leq i \leq k)$ *contains only elements from a set* S_i *of size* $|v_i|$, *2) the rows of each* $N \times t$ *subarray cover all* t*-tuples of values from the* t *columns at least one time, and 3) all rows are models of* C.

Transforming Neural Networks. The GDVB approach manipulates factors that influence DNN verifier performance to construct a diverse benchmark. For DNN construction, we leverage a recent approach, R4V [47], that given an original DNN and an architectural specification automates the transformation of the DNN and uses distillation [28] to train it to closely match the test accuracy of the original DNN. R4V transformation specifications can be written to change a number of architectural parameters of a network including: the input dimension, the range of values for each input dimension, the number of layers, the number of neurons per layer, the number of convolutional kernels, and the stride and padding of a convolutional layer.

3 Identifying Factors that Influence Verifier Performance

As discussed in Sect. 1 the verification community has acted to create policies that incentivize *diverse* benchmarks. Diversity is desirable in a benchmark because it (a) demonstrates the range of applicability of a verification technology and (b) exposes performance variation within and across verification technologies. Consider, that the SMT competition benchmark selection process seeks to "include equal numbers of satisfiable and unsatisfiable benchmarks at different levels of difficulty" [7]. This is due to the fact that the SMT community understands that the satisfiability or unsatisfiability of a benchmark problem is a factor that influences verifier performance[1].

GDVB seeks to make factors influencing verifier performance explicit and to manipulate them to generate a diverse benchmark. To determine an initial set of factors for DNN verifiers we began with an analysis of the literature, which identified several candidate factors, and then conducted a targeted and exploratory **factor study** to identify whether *manipulating a factor could influence some performance measure of some DNN verifier.* This study only aims to identify such factors and does not seek to characterize the complex relationship between factors and DNN verifier performance; for example, we do not aim to capture a comprehensive set of factors, assess the independence of or relations between factors, or rank factors in terms of their degree of influence. A richer and more detailed factor study might further improve the utility of GDVB, but we leave such a study to future work.

3.1 Potential Factors

Relatively few published papers on DNN verification explicitly discuss the factors that influence performance, but nearly all of them present metrics on the verification problems they solved.

Evaluation results for ReLuplex present data on verifier outcome and solve time for local robustness properties that vary in the input center point and radius [30]; most subsequent papers report similar property variation. Evaluation results for RobustVerifier present a study of varying the number of layers in the DNN and its impact on verifier performance [36]. Evaluation results for ERAN present performance variations across a range of networks varying in the number of layers, layer types, and neurons [22,50–52]. Bunel et al. [14] were the first that we are aware of to explicitly vary factors of DNN verification problems. They found that the performance varied with input dimension, number of neurons per layer, and number of layers across a set of 6 different DNN verifiers. All of the other papers published on DNN verification in recent years have used verification problems that varied, in an ad-hoc fashion, over a subset of the above factors.

[1] Since unsatisfiability requires the consideration of all possible variable assignments which generally is more costly than finding a single satisfiable assignment.

3.2 Exploratory Factor Study

As in other verification domains, DNN verifier performance is multi-faceted. In our study, we consider both verification time and accuracy. We say that the result of a verification problem is *accurate* if a verifier determines conclusively that the problem is *valid* or *invalid*, result as opposed to *unknown*[2].

We study factors associated with both properties and DNNs. Based on the literature analysis, we identified 2 factors related to the correctness property: *scale* and *translation*. Scaling a property involves increasing the size of the input

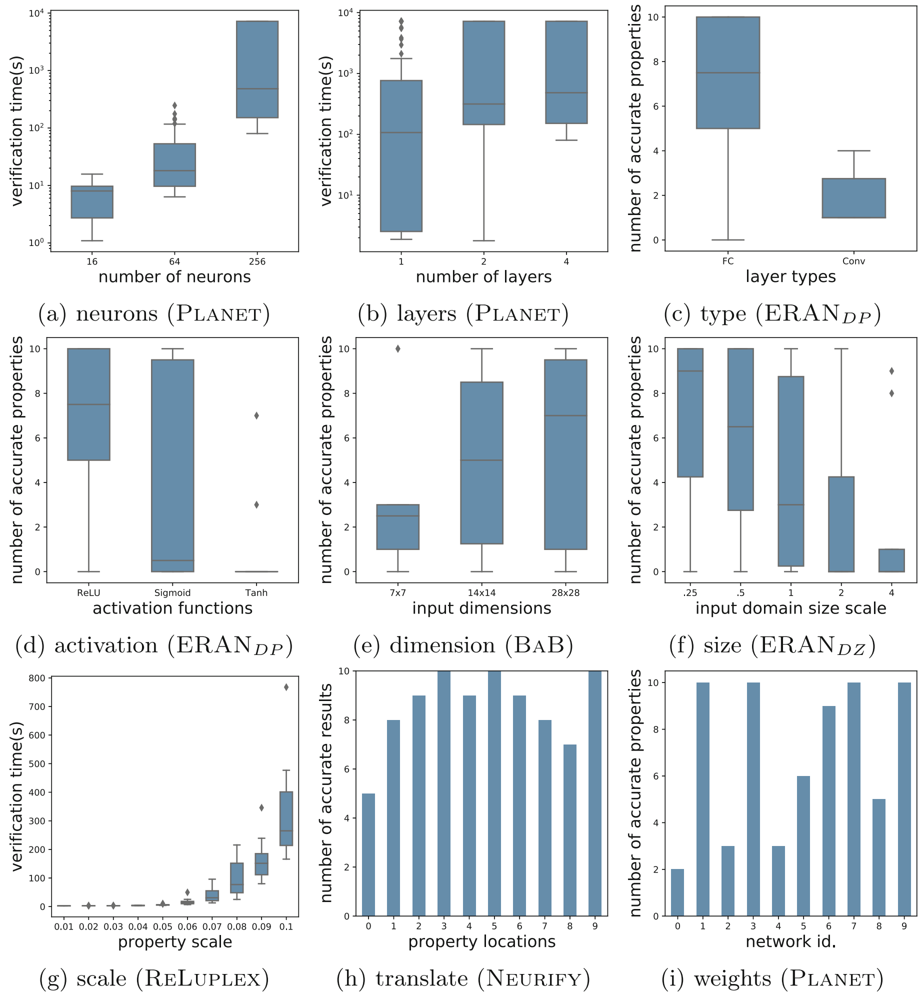


Fig. 1. DNN verifier performance across factors

[2] We cross-check accurate results with multiple verifiers.

domain which will involve *more DNN behavior* in verification. Translating a property involves moving it to a different location in the input domain which will involve *different DNN behavior* in verification. For robustness properties, scaling and translation involve changing the radius and center point of the hypercube describing the input space under verification. One might wonder whether rotation of a property can influence verification performance. For robustness properties, this seems unlikely given their symmetry, but it could be a factor for more irregular input regions – we leave this for future work.

Based on the literature analysis, we identified 4 factors related to the DNN: number of *neurons*, number of *layers*, the *type* of layers, the input *dimension*. We conjectured that an additional 3 factors might impact verifier performance: the type of *activation* function, the input domain *size*, and the learned *weights*.

Our exploratory factor study is opportunistic in that we seek to find a verification problem for which manipulation of a selected factor exhibits performance variation. Towards this end, we conducted a series of trials where we vary a factor hypothesized to influence verification performance, while holding all other factors constant and report the results in Fig. 1. We studied variations of networks for the MNIST task and considered local robustness properties since these were well-supported across a range of different verifiers. We used different verifiers across the study: ReLuplex, Planet, Neurify, BaB, ERAN with the DeepPoly (DP) and DeepZono (DZ) abstract domains. We now briefly describe the trials and then summarize the outcome.

Number of Neurons: The architecture of the DNN was fixed, with 4 fully-connected layers using ReLU activation functions, and the total number of neurons was varied (16, 64, 256) – they were spread evenly across layers. Each network is trained 10 times and verified on 100 local robustness properties. Figure 1(a) plots the number of neurons versus verification time for Planet. **Verification time can increase with the number of neurons.**

Number of Layers: We use the same context as for the neuron factor study, except that we fixed the number of neurons at 256 and vary the number of layers (1, 2, 4). Figure 1(b) plots the number of layers versus verification time for Planet. **Verification time can increase with the number of layers.**

Layer Types: We use a pair of two-layer neural networks, with the same number of neurons, where one has a fully-connected layer and the other a convolutional layer. Each network is trained 10 times and verified on 10 local robustness properties. Figure 1(c) plots layer type versus the number of properties for which accurate results are produced using ERAN_{DP}. **Verification accuracy can vary with layer type.**

Activation Function: We use the fully-connected network from the layer types study, we generated three networks by altering the activation function to use sigmoid and tanh. The training setup and properties remain the same as in the previous trial. Figure 1(d) plots the activation function versus the number of properties for which accurate results are produced using ERAN_{DP}. **Verification accuracy can vary with activation function.**

Input Dimension: We use 3 architectures that differ only in their input dimension which is scaled $(\frac{1}{16}, \frac{1}{4}, 1)$ relative on the original problem. The training setup and properties are from the layer type study. Figure 1(e) plots the input dimension versus the number of properties for which accurate results are produced using BaB. **Verification accuracy can increase with increasing input dimension.**

Input Size: We use 5 architectures that differ only in the range of values of their inputs which are scaled ($\frac{1}{4}, \frac{1}{2}, 1, 2, 4$) based on the original problem. The training setup and properties are from the layer type study. Figure 1(f) plots the input size versus the number of properties for which accurate results are produced using ERAN$_{DZ}$. **Verification accuracy can decrease with increasing input domain size.**

Property Scale: We use a single-layer network and reuse the training setup and properties from the layer type study. We scale the properties ($0.01 - 0.1$) to generate verification problems. Figure 1(g) plots property scaling versus the verification time using ReLuplex. **Verification time can increase with increasing property scale.**

Property Translation: We replicated the property scale study, but held the scale fixed and translated the center point of the local robustness property to 10 other locations. Figure 1(h) plots the number of DNNs for each of the 10 translated properties for which accurate results could be produced using Neurify. **Verification accuracy can vary with property translation.**

Network Weights: Building of the property studies, we explore the verification of 10 scaled property variants across the same network trained 10 times with different initial weights. Figure 1(i) plots the number of accurate properties for which the results could be produced using Planet. **Verification accuracy can vary with the learned weights of the network.**

Exploraty Study Findings. Varying the factors studied influences the performance of different DNN verifiers differently – in terms of time or accuracy. For example, we found that: varying input dimension impacts BaB's accuracy, but not ReLuplex's; varying input domain size impacts ERAN$_{DZ}$'s accuracy, but not Neurify's; and varying property scale impacts ReLuplex's verification time, but not Neurify's.

This study provides a starting set of viable factors that can be used to parameterize the GDVB approach to produce verification problem benchmarks in which those factors are systematically varied. Futhermore, as we discuss in Sect. 4, GDVB generative process allows for us to accommodate information about new factors that might be revealed in future factor studies.

4 The GDVB Approach

The goal of GDVB is to meet requirements R1–R3 by producing a *factor diverse* benchmark that (a) reflects aspects of the complexity encoded in a real verification problem that acts as a seed for generation $\langle n_s, \phi_s \rangle$, (b) varies aspects of the problem that are related to verifier performance, (c) accounts for interactions among those factors, and (d) is comprised only of well-defined verification problems.

Rather than synthesize random verification problems, we seed the generation process in order to generate a benchmark that reflects the complexity of the seed problem. This permits benchmarks to be generated to reflect the challenges present in different DNN problem sub-domains.

Factors, like those described in Sect. 3, may interact; changes to one factor may mask or amplify DNN verifier performance changes arising from another. Exploring all combinations of factors is expensive, but by using covering arrays we can systematically explore interactions among factors. Accounting for such interactions helps to produce a benchmark that is *less biased* than one that only covers individual factor variations.

Not all combinations of factors are possible. For example, if one reduces the number of layers in a network to 0, then it is not possible to preserve the number of neurons in the original network. Thus, benchmark generation must take into account constraints among factors to ensure that only well-defined problems are included in a benchmark.

4.1 Factor Diverse Benchmarks

Consider a set of factors, F, with a set of levels, L_f, for each factor, $f \in F$; we refer to L_f as the *level set* of f. For a verification problem, p, let $l(p)$ be the set of factor levels corresponding to the problem. A benchmark, B, is a set of verification problems and we can denote the factor levels for the benchmark as $l(B) = \{l(p) \mid p \in B\}$.

The simplest form of diversity for a benchmark is requiring that all individual factor levels be present in at least one verification problem, $\forall f \in F : \forall l \in L_f : \exists p \in l(B) : l \in p$. However, this diversity fails to account for interactions among factors. The simplest form of interaction-sensitive diversity considers pairs of factors, but as we discuss below our approach generalizes to any arity of factor-level coverage.

For a pair of factors, $f, f' \in F$, the Cartesian product of their level sets defines the set of all pairwise combinations of their levels. Across all factors the set of such pairs is $\mathit{pairs}(F) = \{(l, l') \mid f, f' \in F \wedge f \neq f' \wedge l \in L_f \wedge l' \in L_{f'}\}$. A *pairwise diverse benchmark* is one in which

$$\forall (x, y) \in \mathit{pairs}(F) : \exists p \in l(B) : (x, y) \in \{(x', y') \mid x' \in p \wedge y' \in p\}$$

Constraints on allowable combinations of factors serve to restrict a benchmark. A pairwise exclusion constraint, $\gamma(F) \subseteq \mathit{pairs}(F)$, requires that

$$\forall (x, y) \in \gamma(F) : \forall p \in l(B) : \neg(x \in p \wedge y \in p)$$

We write γ when F is understood from the context.

The arity of factor-level coverage and exclusion constraints can vary independently. It is common for factor-level coverage to be uniform and to generalize it to t-way coverage, i.e., to require coverage of the elements of the Cartesian product of the level sets of t factors. On the other hand, as observed in prior work [15], constraints generally involve a mix of arity. To denote this generality we define $\Gamma \subseteq \bigcup_i \gamma_i$ where γ_i defines the set of possible i-way exclusion constraints.

Example. Consider the DAVE-2 DNN which accepts 100 by 100 color images and infers an output indicating the steering angle [12]. DAVE-2 is comprised of 5 convolutional layers with 55296, 17424, 3888, 3136, and 1600 neurons, respectively, followed by 4 fully connected layers with 1164, 100, 50, and 10 neurons, respectively. All 82668 neurons use ReLU activations. One can define a local robustness property for DAVE-2 as

$$\phi = \forall \boldsymbol{x} \in i \pm 0.02 : \|\text{DAVE-2}(\boldsymbol{x}) - \text{DAVE-2}(i)\| \leq 5$$

which states that for a given an input image, i, all inputs within a distance of 0.02 will result in an inferred steering angle within $5°$ of the angle for i. These yield the verification problem $\langle \text{DAVE-2}, \phi \rangle$.

Consider factors for the number of neurons, number of convolutional layers, and number of fully-connected layers; a tuple $(\#neuron, \#conv, \#fc)$ represents levels for these factors. For each factor consider two percentage levels: 100% and 50%. A neuron factor level of 50% indicates that a version of DAVE-2 with 41334 neurons is required. In the absence of constraints, an example pairwise factor diverse benchmark for $\langle \text{DAVE-2}, \phi \rangle$ consists of the following four verification problems: $(100\%, 100\%, 100\%)$, $(100\%, 50\%, 50\%)$, $(50\%, 100\%, 50\%)$, and $(50\%, 50\%, 100\%)$. The property ϕ is constant across the benchmark.

4.2 From Factor Covering Arrays to Verification Problems

Given a set of factors, $F = \{f_1, f_2, \ldots, f_{|F|}\}$, and levels, L_{f_i}, a t-way factor diverse benchmark of k verification problems is specified by

$$CMCA(|F|; t, k, (|L_{f_1}|, |L_{f_2}|, \ldots, |L_{f_{|F|}}|), \Gamma)$$

Each element in this mixed level covering array specifies how to construct a verification problem in the benchmark from the seed problem.

Levels are operationalized as transformations on verification problems. We assume a sufficient set of transformations, Δ, such that a verification problem can be transformed into a form that achieves any level of any factor

$$\forall f \in F : \forall l_f \in L_f : \exists \delta \in \Delta : l_f \in l(\delta(\langle n_s, \phi_s \rangle))$$

The definition of Δ and L_i must be coordinated to achieve this property.

A per-factor transformation $\delta \in \Delta$ may impact a single component of a verification problem, e.g., reducing the number of neurons in a DNN does not impact the property, or both components, e.g., the input dimension impacts the DNN and the property by transforming the input data domain. The set of all transformations Δ defines the set of verification problems that can be produced by application of a set of per-factor transformations to the seed problem,

$$\Delta(\langle n_s, \phi_s \rangle) = \{\langle n, \phi \rangle \mid \langle n, \phi \rangle = \delta_{f_1} \circ \delta_{f_2} \ldots \circ \delta_{f_{|F|}}(\langle n_s, \phi_s \rangle) \wedge \delta_i \in \Delta\}$$

The set of all possible factor level combinations is $\Pi_{f \in F} L_f$, i.e., the product of all of the per-factor levels. The set of t-way factor level combinations is

$$c_t = \{c | a \in \Pi_{f \in F} L_f \wedge c \subseteq a \wedge |c| = t\}$$

allowing for the interpretation of $|F|$-tuples as sets.

Definition 3. *Given a set of factors F, with associated factor levels L_f, a t-way factor diverse benchmark, B, for a seed problem $\langle n_s, \phi_s \rangle$ with exclusion constraints Γ is defined by the following: (1) $B \subseteq \Delta(\langle n_s, \phi_s \rangle)$; (2) $\forall \langle n, \phi \rangle \in B : \forall \gamma \in \Gamma : \gamma \not\subseteq l(\langle n, \phi \rangle)$; and (3) $\forall c \in c_t - \Gamma : \exists \langle n, \phi \rangle \in B : c \subseteq l(\langle n, \phi \rangle)$*

4.3 Generating Benchmarks

GDVB is defined in Algorithm 1. We use existing techniques, e.g. Automated Combinatorial Testing for Software (ACTS) [34], for generating a CMCA for constraints specified as logical formulae where factors are variables and levels are values for those variables. A CMCA is a set of k-tuples. Each such tuple defines the target level for each factor for a problem in the generated benchmark. Those levels are used to transform the given seed verification problem and the resultant problem is accumulated in the benchmark.

Algorithm 1: GDVB($\langle n_s, \phi_s \rangle, F, \Gamma, t$) Algorithm

```
Data: a seed problem $\langle n_s, \phi_s \rangle$, a set of factors $F$ and constraints $\Gamma$, a coverage
      goal $t$
Result: A benchmark of DNN verification problems $B$
1 $C \leftarrow$ genCMCA($F, \Gamma, t$)
2 $B \leftarrow \emptyset$
3 for $c \in C$ do
4     $B \leftarrow B \cup$ transform($\langle n_s, \phi_s \rangle, c$)
5 end
```

transform uses different approaches to transform the seed DNN and the property. DNN transformation builds on an approach called R4V that automates architectural transformations to DNNs by scaling (1) the number of neurons in a fully connected layer, (2) the number of kernels in a convolutional layer, (3) the input dimension, or (4) the range of values within an input dimension [47]. The first 3 of these require changes to the structure of the DNN and the last two require changes to the training data, e.g., reshaping, renormalizing. R4V ensures that the network is well-defined after transformation. transform maps factor-levels to per-layer scale parameters for R4V.

R4V permits the training of a network using network distillation which we find advantageous for GDVB because: it accelerates the training process, and it drives training to match the accuracy of the problem DNN to that of n_s, which reduces variation in accuracy across B. We adapt R4V so that after each training epoch, the learned DNN weights and the validation accuracy is recorded. When training finishes, we select the weights associated with the highest validation accuracy. Training is performed using the training data and hyperparameters for n_s.

Whereas R4V can be used to directly manipulate DNN architecture related factors, it can only indirectly affect the learned weights. To address this, we adopt the approach taken throughout the machine learning literature – train a network on multiple initial seeds and report performance across seeds. Thus, each DNN in B is trained multiple times, thereby producing a benchmark comprised of $s * |B|$ verification problems, where is the desired number of seeds.

DNN Transformation Example. Consider this element of the CMCA described above: $\langle(50\%, 100\%, 50\%), \phi\rangle$, applied to DAVE-2. TRANSFORM would compute that 50% of the fully connected layers should be present in the resultant DNN and randomly select 2 of the 4 layers to scale by 0. The fully-connected layers are chosen at random, since the layer count factor does not consider layer ordering. If we consider the case where the layers with 100 and 50 neurons are dropped, this will eliminate 150 neurons. The other transformation required is to reduce the number of neurons by half. To do that all remaining layers will be scaled by $\frac{82668 * 0.5 - 150}{82688} = 0.498$.

Property transformation builds on a domain-specific language (DSL) for specifying DNN correctness properties defined by the *deep neural network verification framework* (DNNV) [48]. Specifications in this Python-based DSL are parametric and TRANSFORM maps factor-levels to those parameters. For example, Fig. 2 defines the parametric local robustness property ϕ that is centered at the image stored at "path/to/image", has radius 0.02, and can be translated and scaled through parameters `t` and `s`, respectively.

```
N=Network("N")
s=Parameter("s",float,
            default=1.0)
e=0.02*s
x=Image("path/to/image")
t=np.load(Parameter("t",
    str,
    "path/to/zeros.npy"))
x=x+t
Forall(x_,
 Implies(
  (x-e)<x_<(x+e),
  abs(N(x_)-N(x)) <= 5
)
```

Fig. 2. Parametric property ϕ

Restricting factors to levels that are supported by TRANSFORM and using CMCA algorithms that meet Definition 2 ensures that GDVB produces a solution that meets Definition 3.

4.4 An Instantiation of GDVB

We developed an instance of GDVB[3] that supports a set of factors informed by the results of the study in Sect. 3, *percentage-based levels* for those factors, and a

[3] https://github.com/edwardxu0/GDVB.

set of constraints that restrict benchmark problems to those that are non-trivial and that can be efficiently trained.

Our instantiation of GDVB supports the following factors: the total number of neurons in the DNN (**neu**), the number of fully-connected layers (**fc**), the number of convolutional layers (**conv**), the dimension of the DNN input (**idm**), the size of each DNN input dimension (**ids**), the scale of the property (**scl**), and the translation of the property (**trn**). We do not support an activation function factor because only ERAN support non-ReLU activations and, thus, using them would render other verifiers inapplicable for large portions generated benchmarks.

We use quintile factor levels, {20%, 40%, 60%, 80%, 100%}, for factors neu, idm, ids, and scl. To permit the elimination of layer types we extend these levels with an additional quintile, 0%, for fc and conv. For trn, we select a set of five translations that shift the property to be centered on a different instance of the training data; unlike the above levels this level is unordered.

Our instantiation of GDVB exclusion constraints for DAVE-2 are as follows: (1) $fc = 0 \wedge conv = 0$, (2) $conv = 0 \wedge neu \geq 20$, (3) $conv = 0 \wedge idm \geq 80$, and (4) $conv = 100 \wedge idm = 20$. The first of these requires that some layer be present. The second and third are related to the blowup in the size of fully-connected layers that results from dropping all convolutional layers which makes training difficult; limiting the total number of neurons and the reduction input dimension mitigates this. The fourth constraint ensures that the input dimension reduction results in a meaningful network; without it the dimensionality reduction achieved by sequences of convolutional layers yields an invalid network, i.e., the input to some layer is smaller than the kernel size.

These constraints were developed iteratively based on feedback from the R4V tool, which reports when TRANSFORM has specified an invalid DNN, and when training failed to closely approximate the accuracy of the seed network.

We note that this instance of GDVB is flexible in that it permits the customization of levels, as we demonstrate in the next section, to generate a benchmark that focuses on variation in a subset of factors. More generally, GDVB can easily be extended to support additional factors and levels for which an instance of TRANSFORM can be defined. We expect that GDVB will evolve in this way as studies of DNN verifiers are performed.

5 GDVB in Use

In this section we showcase the potential uses of GDVB across a series of artifacts and verifiers, while highlighting the challenges it helps to systematically address.

5.1 Setup

Our evaluation applies GDVB to two seed networks: $\text{MNIST}_{ConvBig}$ and DAVE-2. We selected $\text{MNIST}_{ConvBig}$ because it is one of the largest networks in

ERAN's evaluation [50]; it includes 4 convolutional layers and 3 fully connected layers with 48,074 neurons and 1,974,762 parameters. We selected DAVE-2 to illustrate the application of GDVB to a larger network that has been the subject of other DNN analysis [42]; it has 5 convolutional layers and 5 fully connected layers with 82,669 neurons and 2,116,983 parameters.

Table 1 lists the 9 verifiers we selected for our study. This list includes the most well-known verifiers and verification algorithms. We also select variations of some verification approaches. We use Branch-and-Bound (BaB), as well as a variation of Branch-and-Bound with Smart-Branching (BaBSB). Additionally, we evaluate the ERAN verifier with 4 available abstract domains: DeepZono (ERAN_{DZ}), DeepPoly (ERAN_{DP}), RefineZono (ERAN_{RZ}), and RefinePoly (ERAN_{RP}).

Table 1. Verifiers used in GDVB study

Verifier	Algorithm
Reluplex [30]	Search-optimization
Planet [20]	Search-optimization
BaB [14]	Search-optimization
BaBSB [14]	Search-optimization
Neurify[a] [59]	Optimization
ERAN_{DZ} [50]	Reachability
ERAN_{DP} [51]	Reachability
ERAN_{RZ} [52]	Reachability
ERAN_{RP} [49]	Reachability

[a]We use the version of Neurify provided in DNNV [48], which is modified to be applicable to a wide range of problems, whereas the original version was hard-coded to a particular verification problem [59].

To evaluate verifier performance, we use the *solution-count ranking* (SCR) [57], which counts the number of properties that returned accurate verification results. Additionally, we measured the *penalized average runtime* (PAR2) [6], which is computed as the sum of the verification times for *sat* and *unsat* results and twice time limit for all other verification results.

Table 2. Mean & variance of SCR and PAR2 scores across benchmarks. (The darker and lighter gray boxes indicate the best and second best results.)

	$\text{MNIST}_{ConvBig}$		DAVE-2	
Verifier	SCR	PAR2	SCR	PAR2
ERAN_{DZ}	11.40 ± 0.49	$18,126.80 \pm 488.27$	7.20 ± 1.94	$24,496.20 \pm 1,176.59$
ERAN_{DP}	21.00 ± 0.89	$9,206.00 \pm 806.70$	18.40 ± 2.15	$17,443.00 \pm 1,344.65$
ERAN_{RZ}	10.20 ± 0.40	$19,252.60 \pm 343.66$	5.80 ± 2.14	$25,236.60 \pm 1,253.90$
ERAN_{RP}	12.60 ± 1.02	$16,981.40 \pm 930.71$	10.20 ± 1.83	$22,250.60 \pm 1,186.44$
Neurify	22.00 ± 1.10	$8,636.20 \pm 1,008.63$	19.20 ± 2.56	$17,247.80 \pm 1,397.05$
Planet	7.00 ± 0.63	$23,145.60 \pm 468.18$	3.40 ± 1.62	$27,268.60 \pm 775.56$
BaB	0.20 ± 0.40	$28,689.80 \pm 220.40$	0.00 ± 0.00	$28,800.00 \pm 0.00$
BaBSB	0.00 ± 0.00	$28,800.00 \pm 0.00$	0.00 ± 0.00	$28,800.00 \pm 0.00$
Reluplex	3.20 ± 0.40	$25,757.80 \pm 381.40$	4.40 ± 1.02	$26,023.60 \pm 635.90$

All training and verification took place under CentOS Linux 7. R4V transformation and distillation jobs ran on NVIDIA 1080Ti GPUs. Verification jobs were limited to 4 h and ran on 2.3 GHz and 2.2 GHz Xeon processors with 64 GB of memory, for DAVE-2 and MNIST$_{ConvBig}$, respectively.

5.2 Comparing Verifiers Across a Range of Challenges

Consider the use case where a researcher is attempting to compare a new verifier (e.g., a new algorithm, a revised implementation, an extension to an existing approach) against existing verifiers. As shown earlier, for such comparison to be meaningful, many factors must be considered and properly explored. Given a seed network, a property, a set of factors, and a coverage goal, GDVB can generate a benchmark that helps to reduce bias in conducting such an evaluation.

For this use case we consider seed networks and local robustness properties similar to those from the ERAN$_{DZ}$ study [50] for the MNIST$_{ConvBig}$ verification problem and local robustness properties based on those from the Neurify study [59] for the DAVE-2 verification problem. We run an instance of GDVB using the factors and levels described in Sect. 4.4, a coverage strength of 2, and train 5 versions of each network to account for stochastic weight variation. The total time to generate and train GDVB (MNIST$_{ConvBig}$, ...) was 24.3 h and the resulting 30 verification problems took 401.8 h to run across all 9 verifiers. For GDVB (DAVE-2, ...) 44 verification problems were generated with training and verification times of 158.2 h and 772.4 h, respectively. CMCA generation took less than a minute for both problems. Each problem in the benchmark must be trained and verified in sequence, but across problems they can be parallelized. We exploited this to reduce the cost of running the benchmarks to 4.9 h for MNIST$_{ConvBig}$ and 7.9 h for DAVE-2. We measured the SCR and PAR2 score for the nine verifiers across the benchmarks.

The results are shown in Table 2. Since the SCR and PAR2 score trends are the same we depict just SCR in Fig. 3. Boxplots show the SCR scores for a verifier across all the generated problems; variation in plots arises from the 5 trained versions of the networks for each problem. For each box, the middle line represent the median, the box-bounds are the first and third quartiles, and the whiskers represent minimal and maximal values.

The plot for MNIST$_{ConvBig}$ on the left of Fig. 3 shows that **the GDVB benchmark with the MNIST$_{ConvBig}$ seed is able to identify considerable performance variation across verifiers**, with ERAN$_{DP}$ and Neurify accurately verifying a median of over 20 properties, the rest of the ERAN-variants verifying between 10 and 13 properties, and the remaining tools verifying between 0 and 8 properties. The results are consistent when we employ DAVE-2 as the seed network, with **marked differences among groups of verifiers** although the generated problems turned out to be more challenging across all verifiers. ERAN$_{DP}$ and Neurify, the top performers, can verify less than half of the generated problems. Verifiers like BaB were unable to verify any problem derived from DAVE-2 because of the complexity of the seed problem. This point highlights the need for benchmarks to evolve with networks that incorporate emerging technology, and also GDVB's ability to automatically generate a benchmark from different seeds to address that need.

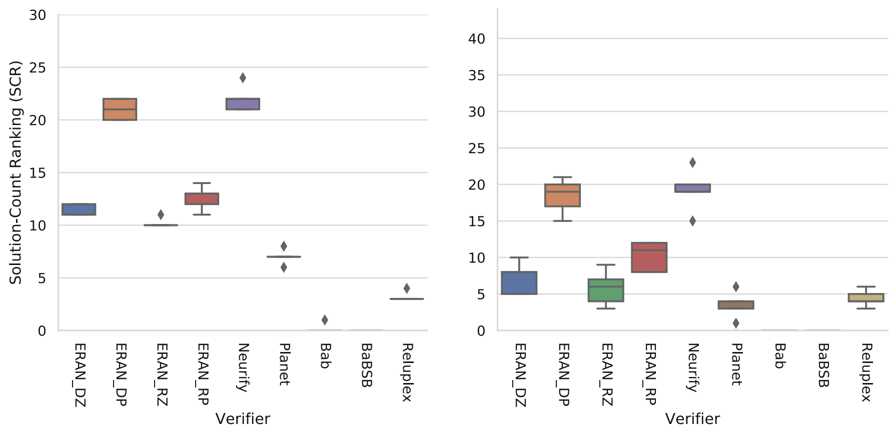


Fig. 3. SCR score for nine verifiers on GDVB benchmarks with MNIST$_{ConvBig}$ (left) and DAVE-2 (right) seeds

Now, understanding the overall performance of a family of verifiers is useful but it is likely just the first step for a researcher to understand under what conditions a verifier excels or struggles. When such conditions correspond to the factors manipulated by GDVB, then they are readily available for further analysis. One analysis may consist of simply plotting the data across its multiple dimensions. We do so in the form of radar-charts for DAVE-2 in Fig. 4 and for MNIST$_{ConvBig}$ in Fig. 5[4]. Since the observations we can gather from both networks are similar, we just discuss DAVE-2 in detail. Each chart includes six axes representing a factor scaled between 0 and 1. The solid lines link the maximum values across factors that were accurately verified while the dotted lines link the median values across factors.

The shape of the lines in the radar plots clearly show that the **verification problems generated by GDVB reveal unique patterns across the verifiers**. For example, the ReLuplex plot indicates that it can do well verifying networks with multiple fully connected (fc) layers but is challenged by larger networks (neu) and those with convolutional layers (conv). Comparing multiple charts also reveals some interesting trade-offs. For example, for smaller networks with just fully connected layers, the medians seem to indicate that ReLuplex is better than Planet. However, when a network incorporates convolutional layers or a larger number of neurons, Planet appears to outperform ReLuplex.

Looking across charts can also pinpoint specific improvements resulting from tool extensions or revisions. For example, the median line of ERAN$_{RZ}$ indicates that it was not as effective in handling verification problems with a larger number of layers as its predecessor ERAN$_{DZ}$; the same trend holds for the pair ERAN$_{RP}$ and ERAN$_{DP}$. We note that a more restrictive benchmark that is biased towards fewer fully connected layers might not reveal such differences.

[4] We do not plot BaBSB as its performance was identical to BaB.

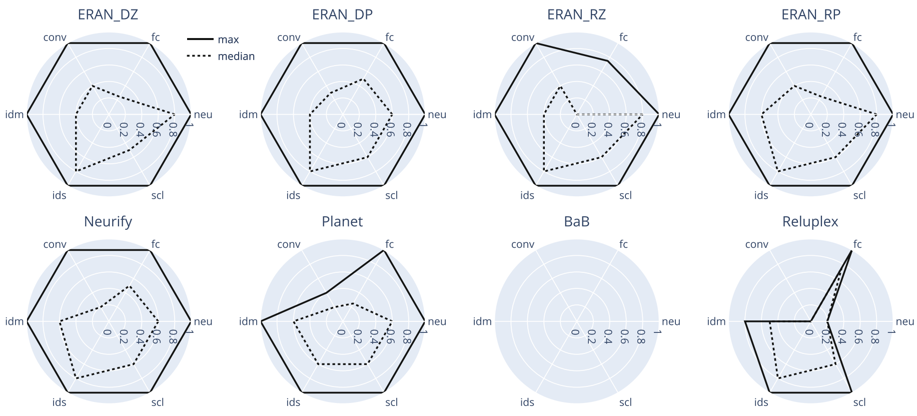


Fig. 4. DAVE-2: radar plot with maximum (solid) and median (dotted) values

GDVB offers the opportunity to investigate such differences even further by generating targeted verification problems for a subset of factors hypothesized to be culprits of those differences. For example, GDVB could generate additional verification problems with a number of fully connected layers between 60% and 80% of the total, while keeping the other factors constant, to refine the understanding of the differences between ERAN_{RZ} and ERAN_{DZ}.

This study illustrates how GDVB benchmarks support the exploration of verifier performance, lowering the burden on researchers to manually prepare tens to hundreds of verification problems, and reducing the opportunities for bias.

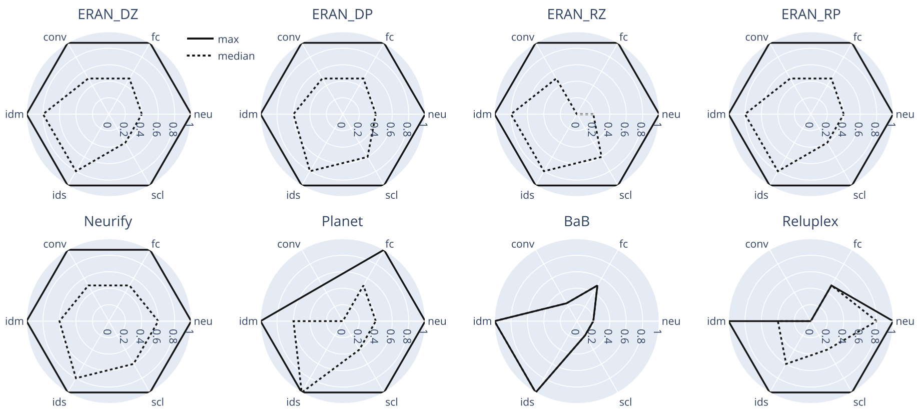


Fig. 5. $\text{MNIST}_{ConvBig}$: radar plot with maximum (solid) and median (dotted) values

5.3 GDVB and Benchmark Requirements R1–R3

As explained in Sect. 1, benchmarking in verification seeks to develop benchmarks that are: diverse; representative of real use cases; and reactive to new technologies. The previous sections have provided evidence of how, through its generative nature, GDVB is reactive to new advances in technology included in the seed network. We have also seen the high degree of parameterization GDVB offers including for setting a seed network from which realistic attributes are inherited in the generated verification problems. In this section we want to illustrate how GDVB addresses the diversity requirement.

To depict diversity we use the parallel coordinate graph in Fig. 6. Each vertical line corresponds to a factor, and the markers in each vertical line corresponds to an explored level. Each verification problem is a polyline that connects the factors' levels explored by it. The two sets of lines correspond to the verification problems included in the DAVE-2 benchmark published with Neurify [59], which is a downsized version of the full DAVE-2 DNN, and the benchmark produced by GDVB (DAVE-2, ...). Each factor in the plot is normalized by dividing by the maximum value for the factor.

Figure 6 shows that the Neurify's DAVE-2 has a large number of neurons, inputs, and dimensions. Yet, it provides very limited coverage of all the factor levels that may affect verification performance. In contrast, GDVB provides a systematic exploration of the factors levels that can affect verifier performance making it much less biased – especially to the numbers of layers in the verification problems, and the combination of those factor levels.

The parallel plot for GDVB benchmark with the MNIST$_{ConvBig}$ seed (not shown for space reasons), depicts a similar trend in terms of systematic exploration of diversity, but since MNIST$_{ConvBig}$ is simpler than DAVE-2, the generated benchmark is correspondingly simpler. This points to the need to identify representative and challenging seeds when parameterizing GDVB. GDVB is fully capable of accomodating factor levels that exceed 100% of a seed network, which is a means of pushing verifiers to the limits of their abilities.

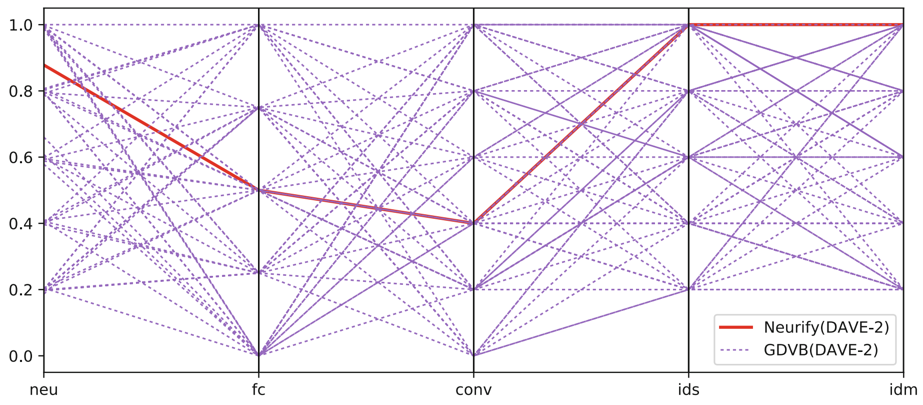


Fig. 6. Diversity explored across factor levels

We note that excluding factors or levels can yield a systematically generated benchmark that is unable to characterize differences between verifiers, or worse, misleads such a characterization by emphasizing certain factors while overlooking others. For example, not exploring different network sizes or exploring networks sizes under 1000 neurons will render similar scores across many DNN verifiers that are differentiated by more comprehensive benchmarks. In applying GDVB, we suggest selecting as many factors as we know may matter, starting from a challenging seed problem, and incrementally refining the levels as needed to focus benchmark results to differentiate verifier performance.

6 Conclusion

The increasing adoption of DNNs has led to a surge in research on DNN verification techniques. Benchmarks to assess these emerging techniques, however, are costly to develop, often lack in diversity and do not represent the population of real evolving DNNs. To address this challenge, we have introduced GDVB, a framework for systematically generating DNN verification problems seeded in complex, real-world networks, ensuring that benchmarks are derived from real problems. GDVB is parameterizable by the factors that may influence verification performance and thereby supports scalable benchmarking. A preliminary study, using 9 DNN verifiers, demonstrates how GDVB can support the assessment of the state-of-the-art.

We plan to conduct broader studies of verifier performance using GDVB, and we encourate other researchers to use and contribute to it. There are many directions to explore in identifying new factors that influence performance, e.g., the impact of quantization and model compression approaches [26]. Work in this direction promises to deepen the community's understanding and lead to advances in DNN verification.

Acknowledgements. This material is based in part upon work supported by the National Science Foundation under grant numbers 1901769 and 1900676, by the U.S. Army Research Office under grant number W911NF-19-1-0054.

References

1. 14th International Satisfiability Modulo Theories Competition. https://smt-comp.github.io/2019/
2. Competition on Software Verification. https://sv-comp.sosy-lab.org/2019/
3. Hardware Model Checking Competition. http://fmv.jku.at/hwmcc19/index.html
4. The International Satisfiability Competitions. http://www.satcompetition.org/
5. Amendola, G., Ricca, F., Truszczynski, M.: A generator of hard 2QBF formulas and ASP programs. In: 16th International Conference on Principles of Knowledge Representation and Reasoning (2018)
6. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT challenge 2012 solver competition. Artif. Intell. **223**, 120–155 (2015)

7. Barrett, C., Deters, M., De Moura, L., Oliveras, A., Stump, A.: 6 years of SMT-COMP. J. Autom. Reasoning **50**(3), 243–277 (2013)
8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0, vol. 13, p. 14 (2010)
9. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Proceedings of the 30th International Conference on Neural Information Processing Systems, pp. 2621–2629 (2016)
10. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019)
11. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 169–190 (2006)
12. Bojarski, M., et al.: End to end learning for self-driving cars. CoRR (2016)
13. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: CNN-Cert: an efficient framework for certifying robustness of convolutional neural networks. In: Association for the Advancement of Artificial Intelligence, January 2019
14. Bunel, R., Turkaslan, I., Torr, P.H., Kohli, P., Kumar, M.P.: A unified view of piece-wise linear neural network verification. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 4795–4804 (2018)
15. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Trans. Softw. Eng. **34**(5), 633–650 (2008)
16. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: 25th International Conference on Software Engineering, pp. 38–48, May 2003
17. D'silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
18. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NASA Formal Methods Symposium, pp. 121–138 (2018)
19. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., Kohli, P.: A dual approach to scalable verification of deep networks. In: Proceedings of the 34th Conference Annual Conference on Uncertainty in Artificial Intelligence, pp. 162–171 (2018)
20. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
21. Elffers, J., Giráldez-Cru, J., Gocht, S., Nordström, J., Simon, L.: Seeking practical CDCL insights from theoretical SAT benchmarks. In: International Joint Conferences on Artificial Intelligence, pp. 1300–1308 (2018)
22. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy, pp. 3–18, May 2018
23. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning, vol. 1. MIT Press, Cambridge (2016)
24. Gopinath, D., Converse, H., Pasareanu, C.S., Taly, A.: Property inference for deep neural networks. In: 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 797–809 (2019)

25. Gupta, A.: Formal hardware verification methods: a survey. In: Kurshan, R. (ed.) Computer Aided Verification, pp. 5–92. Springer, Boston (1992). https://doi.org/10.1007/978-1-4615-3556-0_2
26. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding (2015)
27. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
28. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. In: NIPS Deep Learning and Representation Learning Workshop (2015)
29. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
30. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
31. Katz, G., et al.: The Marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
32. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25: Proceedings of the 26th Annual Conference on Neural Information Processing Systems 2012, pp. 1106–1114 (2012)
33. Kropf, T.: Benchmark-circuits for hardware-verification. In: Kumar, R., Kropf, T. (eds.) TPCD 1994. LNCS, vol. 901, pp. 1–12. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59047-1_39
34. Kuhn, R., Kacker, R.: Automated Combinatorial Testing for Software. https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software
35. Lauria, M., Elffers, J., Nordström, J., Vinyals, M.: CNFgen: a generator of crafted benchmarks. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 464–473. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_30
36. Lin, W., et al.: Robustness verification of classification deep neural networks via linear programming. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 11418–11427 (2019)
37. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. CoRR (2019)
38. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. CoRR (2017)
39. Loquercio, A., Maqueda, A.I., Blanco, C.R.D., Scaramuzza, D.: DroNet: learning to fly by driving. IEEE Robot. Autom. Lett. **3**, 1088–1095 (2018)
40. Meadows, C.A., Meadows, C.A.: Formal verification of cryptographic protocols: a survey. In: Pieprzyk, J., Safavi-Naini, R. (eds.) ASIACRYPT 1994. LNCS, vol. 917, pp. 133–150. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0000430
41. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
42. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 1–18 (2017)

43. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_17
44. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
45. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. In: The International Conference on Learning Representations (2018)
46. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: International Joint Conferences on Artificial Intelligence, pp. 2651–2659 (2018)
47. Shriver, D., Xu, D., Elbaum, S.G., Dwyer, M.B.: Refactoring neural networks for verification. CoRR (2019)
48. Shriver, D.L.: Deep Neural Network Verification Toolbox. https://github.com/dlshriver/DNNV
49. Singh, G., Ganvir, R., Püschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 15072–15083 (2019)
50. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 10802–10813 (2018)
51. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**, article no. 41 (2019)
52. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: Proceedings of the International Conference on Learning Representations (2019)
53. Smolyanskiy, N., Kamenev, A., Smith, J., Birchfield, S.: Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 4241–4247, September 2017
54. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reasoning **43**(4), 337–362 (2009)
55. Sutcliffe, G., Suttner, C.: The TPTP problem library. J. Autom. Reasoning **21**(2), 177–203 (1998)
56. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: International Conference on Learning Representations (2019)
57. Gelder, A.: Careful ranking of multiple solvers with timeouts and ties. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 317–328. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_25
58. Wang, F.: Formal verification of timed systems: a survey and perspective. Proc. IEEE **92**(8), 1283–1305 (2004)
59. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems, pp. 6367–6377 (2018)
60. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security Symposium, pp. 1599–1614 (2018)

61. Weng, T., et al.: Towards fast computation of certified robustness for ReLU networks. In: International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 80, pp. 5273–5282 (2018)
62. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 80, pp. 5283–5292 (2018)
63. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)
64. You, J., Wu, H., Barrett, C., Ramanujan, R., Leskovec, J.: G2SAT: learning to generate SAT formulas. In: Advances in Neural Information Processing Systems, pp. 10552–10563 (2019)

Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI

Daniel J. Fremont[1,2(✉)], Johnathan Chiu[2], Dragos D. Margineantu[3], Denis Osipychev[3], and Sanjit A. Seshia[2]

[1] University of California, Santa Cruz, USA
dfremont@ucsc.edu
[2] University of California, Berkeley, USA
[3] Boeing Research & Technology, Seattle, USA

Abstract. We demonstrate a unified approach to rigorous design of safety-critical autonomous systems using the VerifAI toolkit for formal analysis of AI-based systems. VerifAI provides an integrated toolchain for tasks spanning the design process, including modeling, falsification, debugging, and ML component retraining. We evaluate all of these applications in an industrial case study on an experimental autonomous aircraft taxiing system developed by Boeing, which uses a neural network to track the centerline of a runway. We define runway scenarios using the Scenic probabilistic programming language, and use them to drive tests in the X-Plane flight simulator. We first perform falsification, automatically finding environment conditions causing the system to violate its specification by deviating significantly from the centerline (or even leaving the runway entirely). Next, we use counterexample analysis to identify distinct failure cases, and confirm their root causes with specialized testing. Finally, we use the results of falsification and debugging to retrain the network, eliminating several failure cases and improving the overall performance of the closed-loop system.

Keywords: Falsification · Automated testing · Debugging · Simulation · Autonomous systems · Machine learning

1 Introduction

The expanding use of machine learning (ML) in safety-critical applications has led to an urgent need for rigorous design methodologies that can ensure the reliability of systems with ML components [15,17]. Such a methodology would need to provide tools for *modeling* the system, its requirements, and its environment, *analyzing* a design to find failure cases, *debugging* such cases, and finally *synthesizing* improved designs.

The VerifAI toolkit [1] provides a unified framework for all of these design tasks, based on a simple paradigm: simulation driven by formal models and specifications. The top-level architecture of VerifAI is shown in Fig. 1. We first define an *abstract feature space* describing the environments and system configurations of interest, either by explicitly defining parameter ranges or using the Scenic probabilistic environment modeling language [6]. VerifAI then generates concrete tests by searching this space, using a variety of algorithms ranging from random sampling to global optimization techniques. Finally, we simulate the system for each test, monitoring the satisfaction or violation of a system-level specification; the results of each test are used to guide further search, and any violations are recorded in a table for automated analysis (e.g. clustering) or visualization. This architecture enables a wide range of use cases, including falsification, fuzz testing, debugging, data augmentation, and parameter synthesis; Dreossi et al. [1] demonstrated all of these applications individually through several small case studies.

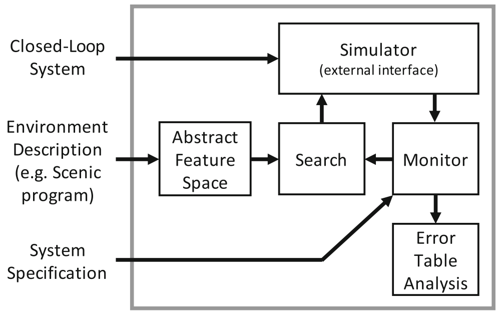


Fig. 1. Architecture of VerifAI.

In this paper, we provide an *integrated* case study, applying VerifAI to a complete design flow for a large, realistic system from industry: TaxiNet, an experimental autonomous aircraft taxiing system developed by Boeing for the DARPA Assured Autonomy project. This system uses a neural network to estimate the aircraft's position from a camera image; a controller then steers the plane to track the centerline of the runway. The main requirement for TaxiNet, provided by Boeing, is that it keep the plane within 1.5 m of the centerline; we formalized this as a specification in Metric Temporal Logic (MTL) [11]. Verifying this specification is difficult, as the neural network must be able to handle the wide range of images resulting from different lighting conditions, changes in runway geometry, and other disturbances such as tire marks on the runway.

Our case study illustrates a complete iteration of the design flow for TaxiNet, analyzing and debugging an existing version of the system to inform an improved design. Specifically, we demonstrate:

1. Modeling the environment of the aircraft using the Scenic language.
2. Falsifying an initial version of TaxiNet, finding environment conditions under which the aircraft significantly deviates from the centerline.
3. Analyzing counterexamples to identify distinct failure cases and diagnose potential root causes.
4. Testing the system in a targeted way to confirm these root causes.
5. Designing a new version of the system by retraining the neural network based on the results of falsification and debugging.
6. Validating that the new system eliminates some of the failure cases in the original system and has higher overall performance.

Following the procedure above, we were able to find several scenarios where TaxiNet exhibited unsafe behavior. For example, we found the system could not properly handle intersections between runways. More interestingly, we found that TaxiNet could get confused when the shadow of the plane was visible, which only occurred during certain times of day and weather conditions. We stress that these types of failure cases are meaningful counterexamples that could easily arise in the real world, unlike pixel-level adversarial examples [8]; we are able to find such cases because VerifAI searches through a space of *semantic* parameters [3]. Furthermore, these counterexamples are *system-level*, demonstrating undesired behavior from the complete system rather than simply its ML component. Finally, our work differs from other works on validation of cyber-physical systems with ML components (e.g. [19]) in that we address a broader range of design tasks (including debugging and retraining as well as testing) and also allow designers to *guide* search by encoding domain knowledge using Scenic.

For our case study, we extend VerifAI in two ways. First, we interface the toolkit to the X-Plane flight simulator [12] in order to run closed-loop simulations of the entire system, with X-Plane rendering the camera images and simulating the aircraft dynamics. More importantly, we extend the Scenic language to allow it to be used in combination with VerifAI's active sampling techniques. Previously, as in any probabilistic programming language, a Scenic program defined a fixed distribution [6]; while adequate for modeling particular scenarios, this is incompatible with active sampling, where we change how tests are generated over time in response to feedback from earlier tests. To reconcile these two approaches, we extend Scenic with *parameters* that are assigned by an external sampler. This allows us to continue to use Scenic's convenient syntax for modeling, while now being able to use not only random sampling but optimization or other algorithms to search the parameter space.

Adding parameters to Scenic enables important new applications. For example, in the design flow we described above, after finding through testing some rare event which causes a failure, we need to generate a dataset of such failures in order to retrain the ML component. Naïvely, we would have to manually write a new Scenic program whose distribution was concentrated on these rare events (as was done in [6]). With parameters, we can simply take the generic Scenic program we used for the initial testing, and use VerifAI's cross-entropy sampler [1,14] to automatically converge to such a distribution [16]. Alternatively, if we have an intuition about where a failure case may lie, we can use Scenic to encode this domain knowledge as a *prior* for cross-entropy sampling, helping the latter to find failures more quickly.

In summary, the novel contributions of this paper are:

- The first demonstration on an industrial case study of an integrated toolchain for falsification, debugging, and retraining of ML-based autonomous systems.
- An interface between VerifAI and the X-Plane flight simulator.

– An extension of the Scenic language with parameters, and a demonstration using it in conjunction with cross-entropy sampling to learn a Scenic program encoding the distribution of failure cases.

We begin in Sect. 2 with a discussion of our extension of Scenic with parameters and our X-Plane interface. Section 3 presents the experimental setup and results of our case study, and we close in Sect. 4 with some conclusions and directions for future work.

2 Extensions of VerifAI

Scenic *with Parameters.* To enable search algorithms other than random sampling to be used with Scenic we extend the language with a concept of *external parameters* assigned by an *external sampler.* A Scenic program can specify an external sampler to use; this sampler will define the allowed types of parameters, which can then be used in the program in place of any distribution. The default external sampler provides access to the VerifAI samplers and defines parameter types corresponding to VerifAI's continuous and discrete ranges. Thus for example one could write a Scenic program which picks the colors of two cars randomly according to some realistic distribution, but chooses the distance between them using VerifAI's Bayesian Optimization sampler.

The semantics of external parameters is simple: when sampling from a Scenic program, the external sampler is first queried to provide values for all the parameters; the program is then equivalent to one without parameters, and can be sampled as usual[1].

X-Plane Interface. Our interface between X-Plane and VerifAI uses the latter's client-server architecture for communicating with simulators. The server runs inside VerifAI, taking each generated feature vector and sending it to the client. The client runs inside X-Plane and calls its APIs to set up and execute the test, reporting back information needed to monitor the specifications. For our client, we used X-Plane Connect [18], an X-Plane plugin providing access to X-Plane's "datarefs". These are named values which represent simulator state, e.g., positions of aircraft and weather conditions. Our interface exposes all datarefs to Scenic, allowing arbitrary distributions to be placed on them. We also set up the Scenic coordinate system to be aligned with the runway, performing the appropriate conversions to set the raw position datarefs.

3 TaxiNet Case Study

3.1 Experimental Setup

TaxiNet's neural network estimates the aircraft's position from a camera image; the camera is mounted on the right wing and faces forward. Example images are

[1] One complication arises because Scenic uses rejection sampling to enforce constraints: if a sample is rejected, what value should be returned to active samplers that expect feedback, e.g. a cross-entropy sampler? By default we return a special value indicating a rejection occurred.

shown in Fig. 2. From such an image, the network estimates the *cross-track error (CTE)*, the left-right offset of the plane from the centerline, and the *heading error (HE)*, the angular offset of the plane from directly down the centerline. These estimates are fed into a handwritten controller which outputs (the equivalent of) a steering angle for the plane.

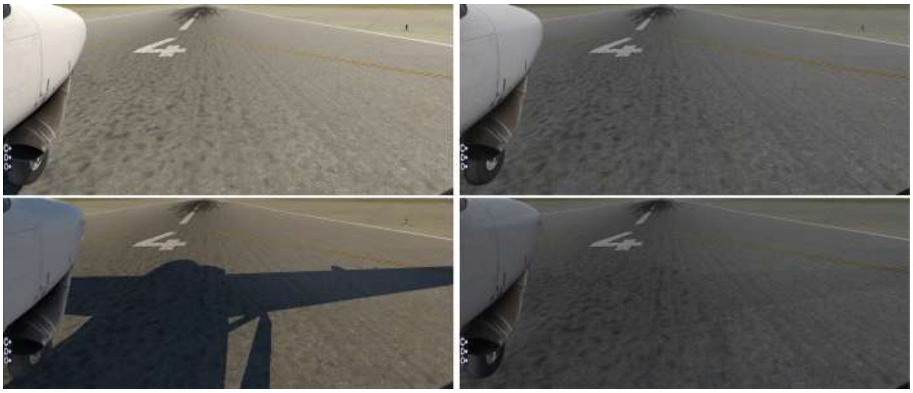


Fig. 2. Example input images to TaxiNet, rendered in X-Plane. Left/right = clear/cloudy weather. Top/bottom = 12 pm/4 pm.

The Boeing team provided the Berkeley team with an initial version of TaxiNet without describing which images were used to train it. In this way, the Berkeley team were not aware in advance of potential gaps in the training set and corresponding potential failure cases[2]. For retraining experiments, the same sizes of training and validation sets were used as for the original model, as well as identical training hyperparameters.

The semantic feature space defined by our Scenic programs and searched by VerifAI was 6-dimensional, made up of the following parameters[3]:

- the initial position and orientation of the aircraft (in 2D, on the runway);
- the type of clouds, out of 6 discrete options ranging from clear to stormy;
- the amount of rain, as a percentage, and
- the time of day.

[2] After drawing conclusions from initial runs of all the experiments, the Berkeley team were informed of the training parameters and trained their own version of TaxiNet locally, repeating the experiments. This was done in order to ensure that minor differences in the training/testing platforms at Boeing and Berkeley did not affect the results (which was in fact qualitatively the case). All numerical results and graphs use data from this second round of experiments.

[3] We originally had additional parameters controlling the position and appearance of a tire mark superimposed on the runway (using a custom X-Plane plugin to do such rendering), but deleted the tire mark for simplicity after experiments showed its effect on TaxiNet was negligible.

Given values for these parameters from VerifAI, the test protocol we used in all of our experiments was identical: we set up the initial condition described by the parameters, then simulated TaxiNet controlling the plane for 30 s.

The main requirement for TaxiNet provided by Boeing was that it should always track the centerline of the runway to within 1.5 m. For many of our experiments we created a greater variety of test scenarios by allowing the plane to start up to 8 m off of the centerline: in such cases we required that the plane approach within 1.5 m of the centerline within 10 s and then stay there for the remainder of the simulation. We formalized these two specifications as MTL formulas φ_{always} and $\varphi_{\text{eventually}}$ respectively:

$$\varphi_{\text{always}} = \Box(\text{CTE} \leq 1.5) \qquad \varphi_{\text{eventually}} = \Diamond_{[0,10]}\Box(\text{CTE} \leq 1.5)$$

While both of these specifications are true/false properties, VerifAI uses a continuous quantity ρ called the *robustness* of an MTL formula [4]. Its crucial property is that $\rho \geq 0$ when the formula is satisfied, while $\rho \leq 0$ when the formula is violated, so that ρ provides a metric of *how close* the system is to violating the property. The exact definition of ρ is not important here, but as an illustration, for φ_{always} it is (the negation of) the greatest deviation beyond the allowed 1.5 m achieved over the whole simulation.

For additional experimental results, see the Appendix of the full version [5].

3.2 Falsification

In our first experiment, we searched for conditions in the nominal operating regime of TaxiNet which cause it to violate $\varphi_{\text{eventually}}$. To do this, we wrote a Scenic program $\mathcal{S}_{\text{falsif}}$ modeling that regime, shown in Fig. 3. We first place a uniform distribution on time of day between 6 am and 6 pm local time (approximate daylight hours). Next, we determine the weather. Since only some of the cloud types are compatible with rain, we put a joint distribution on them: with probability 2/3, there is no rain, and any cloud type is equally likely; otherwise, there is a uniform amount of rain between 25% and 100%[4], and we allow only cloud types consistent with rain. Finally, we position the plane uniformly up to 8 m left or right of the centerline, up to 2000 m down the runway, and up to 30° off of the centerline. These ranges ensured that (1) the plane began on the runway and stayed on it for the entire simulation when tracking succeeded, and (2) it was always possible to reach the centerline within 10 s and so satisfy $\varphi_{\text{eventually}}$.

[4] The 25% lower bound is because we observed that X-Plane seemed to only render rain at all when the rain fraction was around that value or higher.

```
# Time of day: from 6 am to 6 pm. (+8 to get GMT, as used by X-Plane)
param zulu_time = ((6, 18) + 8) * 60 * 60

# Rain: 1/3 of the time. Clouds: rain requires types 3-5; otherwise 0-5.
clouds_and_rain = Options({
    tuple([Uniform(0, 1, 2, 3, 4, 5), 0]): 2,  # no rain
    tuple([Uniform(3, 4, 5), (0.25, 1)]): 1    # 25% to 100% rain
})
param cloud_type = clouds_and_rain[0], rain_percent = clouds_and_rain[1]

# Plane: up to 8 m left/right, 2000 m down the runway, 30° left/right.
ego = Plane at (-8, 8) @ (0, 2000),
            facing (-30, 30) deg
```

Fig. 3. Generic SCENIC program $\mathcal{S}_{\text{falsif}}$ used for falsification and retraining.

However, it was quite easy to find falsifying initial conditions within this scenario. We simulated over 4,000 runs randomly sampled from $\mathcal{S}_{\text{falsif}}$, and found many counterexamples: in only 55% of the runs did TaxiNet satisfy $\varphi_{\text{eventually}}$, and in 9.1% of runs, the plane left the runway entirely. This showed that TaxiNet's behavior was problematic, but did not explain *why*. To answer that question, we analyzed the data VERIFAI collected during falsification, as we explain next.

3.3 Error Analysis and Debugging

VERIFAI builds a table which stores for each run the point sampled from the abstract feature space and the resulting robustness value ρ (see Sect. 3.1) for the specification. The table is compatible with the *pandas* data science library [13], making visualization easy. While VERIFAI contains algorithms for automatic analysis of the table (e.g., clustering and Principal Component Analysis), we do not use them here since the parameter space was low-dimensional enough to identify failure cases by direct visualization.

We began by plotting TaxiNet's performance as a function of each of the parameters in our falsification scenario. Several parameters had a large impact on performance:

– **Time of day:** Figure 4 plots ρ vs. time of day, each orange dot representing a run during falsification; the red line is their median, using 30-min bins (ignore the blue dots for now). Note the strong time-dependence: for example, TaxiNet works well in the late morning (almost all runs having $\rho > 0$ and so satisfying $\varphi_{\text{eventually}}$) but consistently fails to track the centerline in the early morning.
– **Clouds:** Figure 5 shows the median performance curves (as in Fig. 4) for 3 of X-Plane's cloud types: no clouds, moderate "overcast" clouds, and dark "stratus" clouds. Notice that at 8 am TaxiNet performs much worse with stratus clouds than no clouds, while at 2 pm the situation is reversed. Performance also varies quite irregularly when there are no clouds — we will analyze why this is the case shortly.

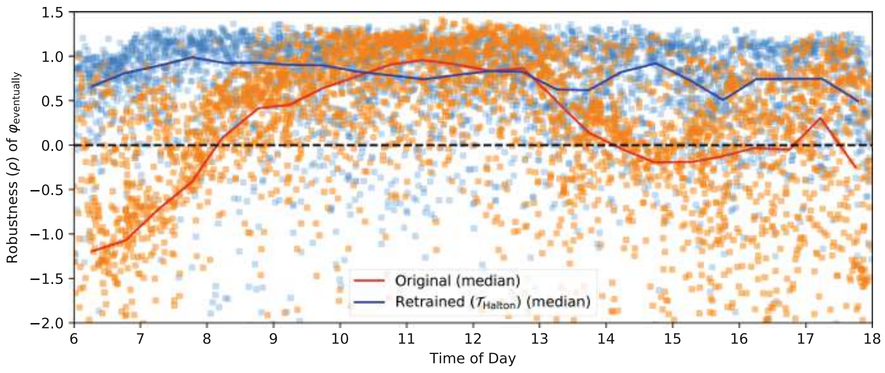


Fig. 4. Performance of TaxiNet as a function of time of day, before and after retraining. (Color figure online)

- **Distance along the runway:** The green data in Fig. 6 show performance as a function of how far down the runway the plane starts (ignore the orange/purple data for now). TaxiNet behaves similarly along the whole length of the runway, except around 1350–1500 m, where it veers completely off of the runway ($\rho \approx -30$). Consulting the airport map, we find that another runway intersects the one we tested with at approximately 1450 m. Images from the simulations show that at this intersection, both the centerline and edge markings of our test runway are obscured.

These visualizations identify several problematic behaviors of TaxiNet: consistently poor performance in the early morning, irregular performance at certain times depending on clouds, and an inability to handle runway intersections. The first and last of these are easy to explain as being due to dim lighting and obscured runway markings. The cloud issue is less clear, but VerifAI can help us to debug it and identify the root cause.

Inspecting Fig. 5 again, observe that performance at 2–3 pm with no clouds is poor. This is surprising, since under these conditions the runway image is bright and clear; the brightness itself is not the problem, since TaxiNet does very well at the brightest time, noon. However, comparing images from a range of times, we noticed another difference: shortly after noon, the plane's shadow enters the frame, and moves across the image over the course of the afternoon. Furthermore, the shadow is far less visible under cloudy conditions (see Fig. 2). Thus, we hypothesized that TaxiNet might be confused by the strong shadows appearing in the afternoon when there are no clouds.

To test this hypothesis, we wrote a new Scenic scenario with no clouds, varying only the time of day; we used VerifAI's Halton sampler [9] to get an even spread of times with relatively few samples. We then ran two experiments: one with our usual test protocol, and one where we disabled the rendering of shadows in X-Plane. The results are shown in Fig. 7: as expected, in the normal run there are strong fluctuations in performance during the afternoon, as the

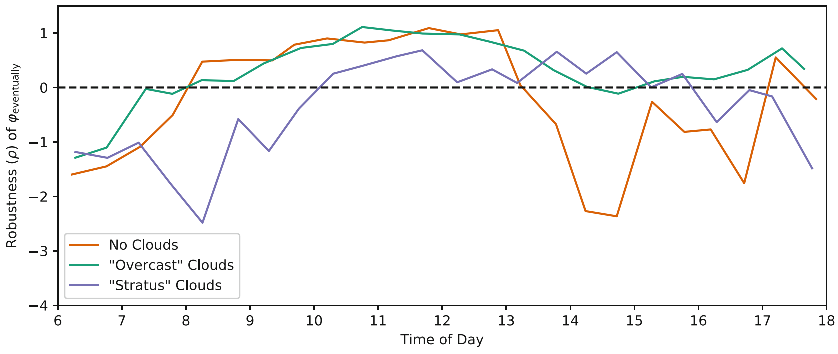


Fig. 5. Median TaxiNet performance by time of day, for different cloud types. (For clarity, individual runs are not shown as dots in this figure.)

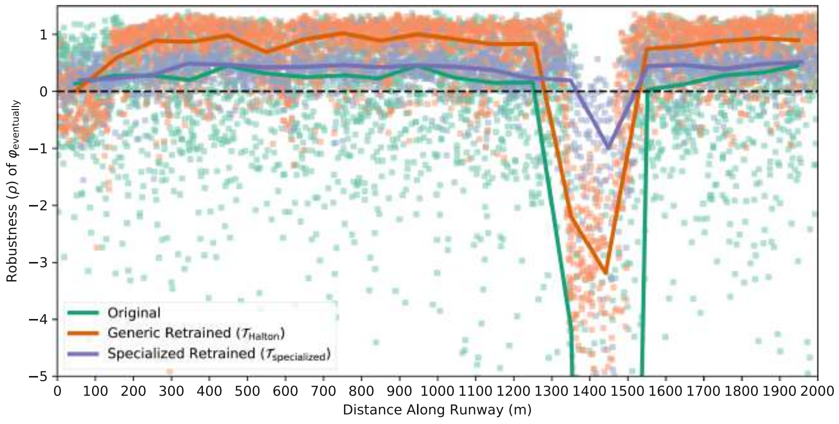


Fig. 6. TaxiNet performance by distance along the runway. Solid lines are medians. The lowest median value for original TaxiNet clipped by the bottom of the chart is −32. (Color figure online)

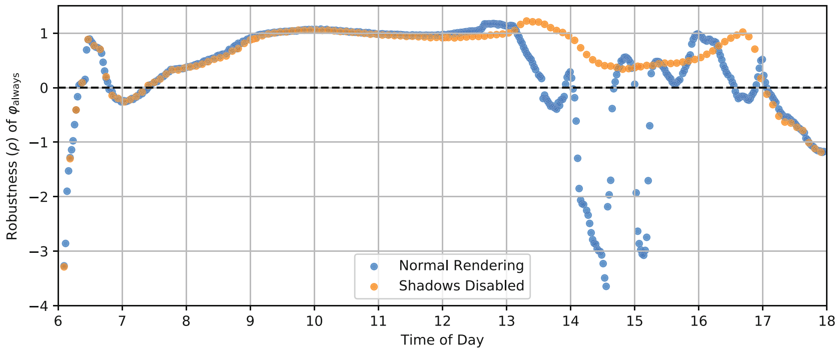


Fig. 7. TaxiNet performance (with fixed plane position) by time of day, with and without shadows.

shadow is moving across the image; with shadows disabled, the fluctuations disappear. This confirms that shadows are a root cause of TaxiNet's irregular performance in the afternoon.

Figures 4 and 6 show that there are failures even at favorable times and runway positions. We diagnosed several additional factors leading to such cases, such as starting at an extreme angle or further away from the centerline; see the Appendix [5] for details.

Finally, we can use VerifAI for fault localization, identifying which part of the system is responsible for an undesired behavior. TaxiNet's main components are the neural network used for perception and the steering controller: we can test which is in error by replacing the network with ground truth CTE and HE values and testing the counterexamples we found above again. Doing this, we found that the system always satisfied $\varphi_{\text{eventually}}$; therefore, all the failure cases were due to mispredictions by the neural network. Next, we use VerifAI to retrain the network and improve its predictions.

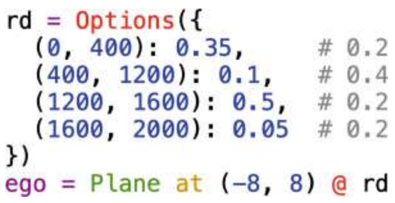
3.4 Retraining

The easiest approach to retraining using VerifAI is simply to generate a new generic training set using the falsification scenario $\mathcal{S}_{\text{falsif}}$ from Fig. 3, which deliberately includes a wide variety of different positions, lighting conditions, and so forth. We sampled new configurations from the scenario, capturing a single image from each, to form new training and validation sets with the same sizes as for original TaxiNet. We used these to train a new version of TaxiNet, $\mathcal{T}_{\text{generic}}$, and evaluated it as in the previous section, obtaining much better overall performance: out of approximately 4,000 runs, 82% satisfied $\varphi_{\text{eventually}}$, and only 3.9% left the runway (compared to 55% and 9.1% before). A variant of $\mathcal{T}_{\text{generic}}$ using VerifAI's Halton sampler, $\mathcal{T}_{\text{Halton}}$, was even more robust, satisfying $\varphi_{\text{eventually}}$ in 83% of runs and leaving the runway in only 0.6% (a 15× improvement over the original model). Furthermore, retraining successfully eliminated the undesired behaviors caused by time-of-day and cloud dependence: the blue data in Fig. 4 shows the retrained model's performance is consistent across the entire day, and in fact this is the case for each cloud type individually.

However, this naïve retraining did not eliminate all failure cases: the orange data in Fig. 6 shows that $\mathcal{T}_{\text{Halton}}$ still does not handle the runway intersection well. To address this issue, we used a second approach to retraining: over-representing the failure cases of interest in the training set using a specialized Scenic scenario [6].

We altered $\mathcal{S}_{\text{falsif}}$ as shown in Fig. 8, increasing the probability of the plane starting 1200–1600 m along the runway, a range which brackets the intersection; we also emphasized the range 0–400 m, since Fig. 6 shows the model also has difficulty at the start of the runway. We trained a specialized model $\mathcal{T}_{\text{specialized}}$ using training data from this scenario together with the validation set from $\mathcal{T}_{\text{generic}}$. The new model had even better overall performance than $\mathcal{T}_{\text{Halton}}$, with 86% of runs satisfying $\varphi_{\text{eventually}}$ and 0.5% leaving the runway. This is because performance near the intersection is significantly improved, as shown by the purple data in Fig. 6; however, while the plane rarely leaves the runway completely, it still typically deviates several meters from the centerline. Furthermore, performance is worse than $\mathcal{T}_{\text{generic}}$ and $\mathcal{T}_{\text{Halton}}$ over the rest of the runway, suggesting that larger training sets might be necessary for further performance improvements.

While in this case it was straightforward to write the Scenic program in Fig. 8 by hand, we can also *learn* such a program automatically: starting from $\mathcal{S}_{\text{falsif}}$ (Fig. 3), we use cross-entropy sampling to move the distribution towards failure cases. Applying this procedure to $\mathcal{T}_{\text{generic}}$ for around 1200 runs, VerifAI indeed converged to a distribution concentrated on failures. For example, the distribution of distances along the runway gave ∼79% probability to the range 1400–1600 m, 16% to 1200–1400 m, and 5% to 0–200, with all other distances getting only ∼1% in total. Referring back to Fig. 6, we see that these ranges exactly pick out where $\mathcal{T}_{\text{Halton}}$ (and $\mathcal{T}_{\text{generic}}$) has the worst performance.



```
rd = Options({
  (0, 400): 0.35,       # 0.2
  (400, 1200): 0.1,     # 0.4
  (1200, 1600): 0.5,    # 0.2
  (1600, 2000): 0.05    # 0.2
})
ego = Plane at (-8, 8) @ rd
```

Fig. 8. Position distribution emphasizing the runway beginning and intersection. Probabilities corresponding to the original scenario (Fig. 3) shown in comments.

Finally, we also experimented with a third approach to retraining, namely augmenting the existing training and validation sets with additional data rather than generating completely new data as we did above. The augmentation data can come from counterexamples from falsification [2], from a handwritten Scenic scenario, or from a failure scenario learned as we saw above. However, we were not able to achieve better performance using such iterative retraining approaches than simply generating a larger training set from scratch, so we defer discussion of these experiments to the Appendix [5].

4 Conclusion

In this paper, we demonstrated VerifAI as an integrated toolchain useful throughout the design process for a realistic, industrial autonomous system. We were able to find multiple failure cases, diagnose them, and in some cases fix them through retraining. We interfaced VerifAI to the X-Plane flight simulator, and extended the Scenic language with external parameters, allowing the combination of probabilistic programming and active sampling techniques. These extensions are publicly available [1,7].

While we were able to improve TaxiNet's rate of satisfying its specification from 55% to 86%, a 14% failure rate is clearly not good enough for a safety-critical system (noting of course that TaxiNet is a simple prototype not intended for deployment). In future work, we plan to explore a variety of ways we might further improve performance, including repeating our falsify-debug-retrain loop (which we only showed a single iteration of), increasing the size of the training set, and choosing a more complex neural network architecture. We also plan to further automate error analysis, building on clustering and other techniques (e.g., [10]) available with VerifAI and Scenic, and to incorporate white-box reasoning techniques to improve the efficiency of search.

Acknowledgments. The authors are grateful to Forrest Laine and Tyler Staudinger for assistance with the experiments and TaxiNet, to Ankush Desai for suggesting using Scenic as a prior for cross-entropy sampling, and to the anonymous reviewers.

This work was supported in part by NSF grants 1545126 (VeHICaL), 1646208, 1739816, and 1837132, the DARPA BRASS (FA8750-16-C0043) and Assured Autonomy programs, Toyota under the iCyPhy center, and Berkeley Deep Drive.

References

1. Dreossi, T., et al.: VerifAI: a toolkit for the formal design and analysis of artificial intelligence-based systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 432–442. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_25
2. Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Counterexample-guided data augmentation. In: 27th International Joint Conference on Artificial Intelligence (IJCAI), pp. 2071–2078, July 2018. https://doi.org/10.24963/ijcai.2018/286
3. Dreossi, T., Jha, S., Seshia, S.A.: Semantic adversarial deep learning. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 3–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_1
4. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification, pp. 178–192. Springer, Berlin (2006)
5. Fremont, D.J., Chiu, J., Margineantu, D.D., Osipychev, D., Seshia, S.A.: Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI (2020). https://arxiv.org/abs/2005.07173
6. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 63–78 (2019). https://doi.org/10.1145/3314221.3314633
7. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation (2019). https://github.com/BerkeleyLearnVerify/Scenic
8. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. CoRR (2014). http://arxiv.org/abs/1412.6572
9. Halton, J.H.: On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numer. Math. **2**(1), 84–90 (1960). https://doi.org/10.1007/BF01386213
10. Kim, E., Gopinath, D., Pasareanu, C.S., Seshia, S.A.: A programmatic and semantic approach to explaining and debugging neural network based object detectors. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2020)
11. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
12. Laminar Research: X-Plane 11 (2019). https://www.x-plane.com/

13. McKinney, W.: Data structures for statistical computing in python. In: van der Walt, S., Millman, J. (eds.) 9th Python in Science Conference, pp. 51–56 (2010). https://pandas.pydata.org/
14. Rubinstein, R.Y., Kroese, D.P.: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning. Springer, New York (2004). https://doi.org/10.1007/978-1-4757-4321-0
15. Russell, S., Dewey, D., Tegmark, M.: Research priorities for robust and beneficial artificial intelligence. AI Mag. **36**(4), 105–114 (2015). https://doi.org/10.1609/aimag.v36i4.2577
16. Sankaranarayanan, S., Fainekos, G.E.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: Hybrid Systems: Computation and Control (part of CPS Week 2012), HSCC 2012, Beijing, China, April 17–19, 2012, pp. 125–134 (2012). https://doi.org/10.1145/2185632.2185653,
17. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards Verified Artificial Intelligence. CoRR (2016). http://arxiv.org/abs/1606.08514
18. Teubert, C., Watkins, J.: The X-Plane Connect Toolbox (2019). https://github.com/nasa/XPlaneConnect
19. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, pp. 303–314. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3180155.3180220

Blockchain and Security

The Move Prover

Jingyi Emma Zhong[1], Kevin Cheang[2], Shaz Qadeer[3], Wolfgang Grieskamp[3], Sam Blackshear[4], Junkil Park[4], Yoni Zohar[1], Clark Barrett[1](✉), and David L. Dill[4]

[1] Stanford University, Stanford, USA
barrett@cs.stanford.edu
[2] UC Berkeley, Berkeley, USA
[3] Novi, Seattle, WA, USA
[4] Novi, Menlo Park, CA, USA

Abstract. The Libra blockchain is designed to store billions of dollars in assets, so the security of code that executes transactions is important. The Libra blockchain has a new language for implementing transactions, called "Move." This paper describes the Move Prover, an automatic formal verification system for Move. We overview the unique features of the Move language and then describe the architecture of the Prover, including the language for formal specification and the translation to the Boogie intermediate verification language.

Keywords: Libra · Blockchain · Smart contracts · Formal verification

1 Introduction

The ability to implement arbitrary transactions on a blockchain via so-called *smart contracts* has led to an explosion in innovative services in systems such as Ethereum [41]. Unfortunately, bugs in smart contracts have led to massive amounts of funds being stolen or made inaccessible [5,15]. In retrospect, the source of these disasters is fairly obvious: smart contracts operate without a safety net. A fundamental requirement for blockchains is that transactions be automatic and irreversible. Unlike traditional financial applications, there is little opportunity for humans to oversee or intervene in transactions. Indeed, the design of the blockchain is intended to prevent human involvement. The resulting potential havoc that can be caused by a bug in a smart contract makes it essential for these contracts to be correct, without vulnerabilities. Not surprisingly, there is great interest in formal verification and other advanced testing methods for smart contracts, and several verification systems already exist or are under development.

This work was supported by the Stanford Center for Blockchain Research and Novi, a Facebook subsidiary whose goal is to provide financial services that let people participate in the Libra network. The Libra Association manages the Libra network and is an independent, not-for-profit membership organization, headquartered in Geneva, Switzerland.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 137–150, 2020.
https://doi.org/10.1007/978-3-030-53288-8_7

The Libra blockchain [3,38] is designed to be a foundation for supporting financial services for billions of people around the world. If successful, it could store and manage assets worth billions of dollars, with correspondingly stringent security requirements. The code that modifies the state of the blockchain is especially important. The architecture of the Libra blockchain requires that all such modifications be performed by the Move [12] virtual machine, which executes the well-defined Move instruction set. This architecture means that verification efforts can focus on the correctness of bytecode programs implementing smart contracts, including formally verifying those programs.

Contributions

In this paper, we describe a specification language and formal verification system for Move. If a programmer writes functional correctness properties for a procedure, the Move Prover tool can automatically verify it. Although many similar Floyd-Hoare verifiers exist, widespread adoption has been a challenge because conventional software is large, complex, and uses language features that present difficulties for even the simplest verification tasks. However, we are hopeful that the Move Prover will be used by the majority of Move programmers. There are three reasons for this optimism. First, the Move language has been designed to support verification. Second, we are building a culture of specification from the beginning: each Move module used by the Libra blockchain is being written with an accompanying formal specification. Finally, we are working to make the Move Prover as precise, fast, and user-friendly as possible.

The Move language, the Move Prover, Move programs, and their specifications, have been evolving rapidly, so this description necessarily represents a snapshot of the project at a particular time. However, we expect most of the changes to be improvements and extensions to the basics described here. In the remainder of this paper, we will:

1. Present a brief overview of Move and explain the language design decisions that facilitate verification (Sect. 2);
2. Describe how the Move Prover toolchain is implemented (Sect. 3);
3. Explain the model used to represent Move programs (Sect. 4);
4. Define the Move specification language and give examples of useful properties it can encode (Sect. 5); and
5. Demonstrate that the Move Prover can verify important aspects of the Libra core modules (Sect. 6).

2 Background: The Move Language

Move [12] is an executable bytecode language for writing smart contracts and custom transaction logic. Contracts in Move are written as *modules* that contain record types and procedures. Records in modules may either be struct or *resource* types—the most novel feature of Move. A resource type has linear [17] semantics, meaning that resources cannot be created, copied, or destroyed except by

```
module LibraCoin {
  resource struct T { value: u64 }

  public fun join(coin: &mut LibraCoin::T, to_consume: LibraCoin::T) {
    let  T { value } = to_consume; // MoveLoc(1); Unpack
    let c_value_ref = &mut coin.value; // MoveLoc(0); MutBorrowField<value>; StLoc(0)
    *c_value_ref = *c_value_ref + value; // CopyLoc(0); ReadRef; Add; MoveLoc(0); WriteRef
    return; // Ret
  }
}
```

Fig. 1. A Move module with its bytecode representation in comments.

procedures in its declaring module. Resources allow programmers to encode safe, yet customizable assets that cannot be accidentally (or intentionally) copied or destroyed by code outside the module.

Move is minimal in comparison to most conventional programming languages. The only types besides records are primitives (Booleans, unsigned integers, addresses), vectors, and references (which must be labeled as mutable or immutable, similar to Rust [30]). Records can contain primitives and other records, but not references. Control-flow constructs can be encoded via jumps to static labels in the bytecode.

Move programs execute in the context of a blockchain with modules and resources published under *account addresses*. To interact with the blockchain, a programmer can write a Move *transaction script*, a single-procedure program similar to a `main` procedure in a conventional language, that invokes procedures of published modules. This script is then packaged into a cryptographically signed transaction that is executed by validators in the Libra blockchain. As in Ethereum, transaction execution is *metered*, meaning that computational resources (or "gas") used when a Move program is executed are measured and must be paid for by the submitter of a transaction (though we note that the Move Prover does not yet reason about gas usage).

Verification-Friendly Design. There are several aspects of Move's design that facilitate verification. The first is limited interaction with the environment: to ensure deterministic execution, the language can only read data from the global blockchain state or the current transaction (no file or network I/O). Second, many features that are challenging for verification are absent from Move: concurrency, higher-order functions, exceptions, sub-typing, and dynamic dispatch. The absence of the last feature is particularly notable because it is present in Ethereum bytecode and has contributed to subtle *re-entrancy* bugs (e.g., [14]). Third, Move has built-in safe arithmetic: overflows and underflows are detected during execution and result in a transaction abort. Finally, many common errors are prevented by the Move *bytecode verifier* (not to be confused with the Move Prover), a static analyzer that checks each bytecode program before execution (similar to the JVM [26] or CLR [31] bytecode verifier). The bytecode verifier ensures that:

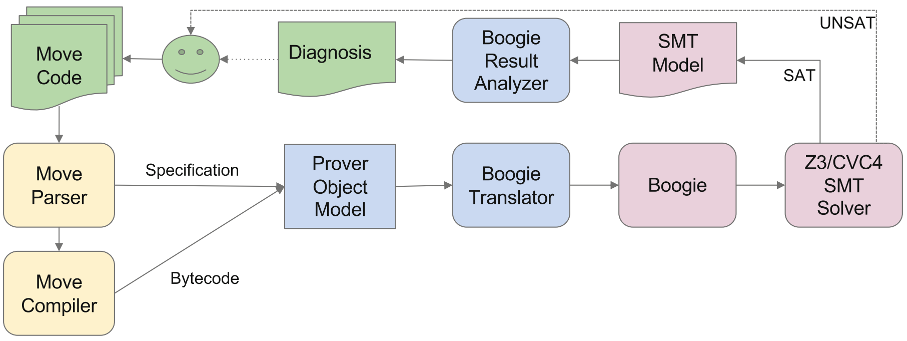


Fig. 2. The Move Prover architecture.

1. Procedures and struct declarations are well-typed (e.g., linearity of resources)
2. Dependent modules and procedure targets exist (i.e., static linking)
3. Module dependencies are acyclic
4. The operand stack height is the same at the beginning and end of each basic block
5. A procedure can only touch stack locations belonging to callers via a reference passed to the callee
6. The global and local memory are always tree-shaped
7. There are no dangling references
8. A mutable reference has exclusive access to its referent

Because these checks are run on every Move bytecode program, the prover can rely on them in its own reasoning. Note that this would not be true if the checks were performed by a source language compiler, since bad bytecode programs could be created by compiler bugs or by writing programs directly in the executable bytecode representation.

Limited Aliasing. In the rest of this section, we present an example that explains the memory-related invariants enforced by the Move bytecode verifier (6–8 above). The example in Fig. 1 is written in the Move source language, which can be directly compiled to the Move bytecode representation shown in the comments (note that the Move Prover analyzes the bytecode itself). The `join` procedure accepts two arguments: `coin` of type `&mut LibraCoin::T` (a mutable reference to a `LibraCoin::T` value stored elsewhere) and `to_consume` of type `LibraCoin::T` (an *owned* `LibraCoin::T` value). The purpose of this procedure is to destroy the `LibraCoin::T` resource stored in `to_consume` and add its value to the `LibraCoin::T` resource referenced by `coin`. The first line of the procedure performs the destruction by "unpacking" `to_consume` (placing the program value bound to its field into the program variable `value`), and the next two lines read the current value of `c_value_ref` and update it.

The careful reader might wonder: what will happen if `c_value_ref` is a reference to `to_consume`? In a C-like language, the first line would make `c_value_ref` into a dangling reference, which would lead to a memory error when it is subsequently used. Fortunately, the Move bytecode verifier ensures that this cannot happen. An owned value like `to_consume` can only be moved (either onto the operand stack or into global storage) if there are no outstanding references to the value. In addition, the bytecode verifier guarantees that no mutable reference can be an ancestor or descendant of another (mutable or immutable) reference in the same local or global data tree. This is a very strong restriction! It ensures that procedure formals that can be mutated (mutable references or owned values) point to disjoint memory locations. For example, an additional formal of type `&mut u64` in the code above could not point into the memory of the other formals. Formals that are immutable references may alias with each other, but not with mutable references or owned values. This means it is impossible for an update to a reference to affect the value retrieved by a simultaneously existing reference. These restrictions on the structure of memory enable greatly simplified reasoning about aliased mutable data, a significant challenge for verification in conventional languages.

3 Tool Overview

Figure 2 shows the architecture of the Move Prover. The prover takes as input Move source code annotated with specifications. The overall workflow consists of several steps. First, the specifications are extracted from the annotated code, and the Move source code is compiled into Move bytecode. Next, all stack operations are removed from the bytecode and replaced with operations on local variables, and the stackless bytecode is abstracted into a prover object model. Along a separate path, the specifications are parsed and added to the prover object model. The finalized model is translated to a program in the Boogie *intermediate verification language (IVL)* [23,24].

The Boogie program is handed to the Boogie verification system, which generates an SMT formula in the SMT-LIB format [10]. This can then be checked using an SMT solver such as Z3 [32] or CVC4 [9]. If the result of this check is UNSAT, then the specification holds, which is reported to the user. Otherwise, a countermodel is obtained from the SMT-solver, which gets translated back to Boogie. Boogie produces a Boogie-level error report, and this result is analyzed and transformed into a source-level diagnosis that is given back to the user. Using this diagnosis, the user can refine the implementation and/or specification and start the process again.

The prover is written in Rust and can be found in the `language/move-prover` directory in the Libra repository on GitHub [25].[1] We describe the Boogie model and the specification language in more detail in the following sections.

[1] This paper reflects the state of the Move Prover at github commit https://github.com/libra/libra/tree/6798b1cd50ac7d524d3e494783910b3d7e827eef.

4 Boogie Model

Boogie IVL is a simple imperative programming language that supports local and global variables, branching and loops, and procedures and procedure calls. Boogie is designed for verification, so it also supports pre- and post-conditions, loop invariants, and global axioms. Boogie programs are not executable; instead, they are provided as input to the Boogie verification system, which applies a verification strategy to generate verification conditions (as SMT formulas) [8]. If all of the verification conditions hold, then each procedure ensures its post-conditions, under the assumption that its pre-conditions hold. The variable types supported by Boogie IVL match the sorts supported by SMT solvers, e.g., Booleans, integers, arrays, bitvectors, and datatypes. This makes the translation of Boogie verification conditions into SMT formulas fairly transparent. Boogie is used as a back-end for a wide variety of verification tools. The general strategy is to model the semantics of a source language in Boogie. Then, programs and specifications in the source language can be translated into Boogie IVL and checked using the Boogie verification system. For more details about Boogie, we refer the reader to [1,7,23,24].

Following this pattern, we built a Boogie model for Move bytecode programs. A few highlights of the model are shown in Fig. 3 and described below. For a detailed understanding of the model, we refer the reader to the full Boogie model, which can be found in the Libra repository at `language/move-prover/src/prelude.bpl` and to a formalization of the core Move bytecode language described in [13].

As mentioned above, in Move, a data value is either a primitive value (e.g., Boolean, integer, address), a struct (i.e. a record) containing one or more data values, or a vector of data values. Data values are represented in Boogie as the `Value` datatype, with one constructor for each primitive type, plus a *vector* constructor (containing one field: a finite array of `Value`), used to model both vectors and structs.

Because Move supports generic functions (i.e. type-parameterized functions), we define a similar Boogie datatype for types called `TypeValue` (not shown). A type-parameterized function can then be represented as a Boogie procedure whose initial arguments are of type `TypeValue` (for the type parameters) and whose data arguments are of type `Value` (regardless of their actual Move type). The bytecode verifier ensures type-correctness, so we do not check that types are used correctly, but rather assume this is the case (by using Boogie `assume` statements as needed).

The `Value` and `ValueArray` datatypes are mutually recursive, and thus a `Value` can be thought of as a finite tree. A primitive `Value` is a leaf node of the tree, while a struct or vector `Value` is an internal node. A position within the tree can be uniquely identified by a *path*, which is a sequence of integers. A path specifies a node of the tree by starting at the root node and then following children according to the indices in the path. We model paths as finite arrays (also shown in Fig. 3). This simplifies the specification that two trees are disjoint, which is a necessary precondition in some smart contract functions.

```
type {:datatype} Value;
function {:constructor} Boolean(b: bool): Value;
function {:constructor} Integer(i: int): Value;
function {:constructor} Address(a: int): Value;
function {:constructor} Vector(v: ValueArray): Value;

type {:datatype} ValueArray;
function {:constructor} ValueArray(v: [int]Value, l: int): ValueArray;

type {:datatype} Path;
function {:constructor} Path(p: [int]int, size: int): Path;

type {:datatype} Location;
function {:constructor} Global(t: TypeValue, a: int): Location;
function {:constructor} Local(i: int): Location;

type {:datatype} Reference;
function {:constructor} Reference(l: Location, p: Path): Reference;

type {:datatype} Memory;
function {:constructor} Memory(domain: [Location]bool, contents: [Location]Value): Memory;
var $m : Memory;
```

Fig. 3. Highlights of the Boogie model for the Move Prover. The `type {:datatype}` syntax is used to declare a new datatype, and the `function {:constructor}` syntax is used to declare datatype constructors with their selectors. An array indexed by type `T` containing elements of type `V` is denoted in Boogie as `[T]V`.

A `Value` can be stored in either local or global state, and references to data in either are allowed as local variables. For simplicity and uniformity, we have a single memory object which is a map from `Location` to `Value` (because memory is a partial function, it also contains a map from `Location` to `bool`, which indicates whether a particular location is present in memory). A `Location` is either global (indexed by an account address and a type) or local (indexed by an integer). References are then represented as a pair consisting of a location and a path. To model reading from or writing to a reference, the global memory is accessed along the reference's path. Note that this is done by enumerating cases up to the maximum possible path depth (based on the data structures in the modules being verified).[2]

Finally, each bytecode instruction is modeled as a procedure modifying local or global state in Boogie. A bytecode program is then translated to a sequence of procedure calls, with `goto` statements handling control-flow.

[2] As with most verification approaches based on generating verification conditions, verifying recursive procedures or loops in Boogie requires writing loop invariants, which can be difficult and may also introduce quantifiers, making the problem harder for the underlying SMT solver. We have avoided this so far by relying on bounded iteration, but our roadmap includes full handling of recursion and loops via loop invariants.

```
public fun pay_from_sender(payee: address, amount: u64) acquires T
{
  Transaction::assert(payee != Transaction::sender(), 1);  // new!

  if (!exists<T>(payee)) {
    Self::create_account(payee);
  };
  Self::deposit(
    payee,
    Self::withdraw_from_sender(amount),
  );
}

spec fun pay_from_sender {
//  ... omitted aborts_ifs ...
  aborts_if amount == 0;
  aborts_if global<T>(sender()).balance.value < amount;
  ensures exists<T>(payee);
  ensures global<T>(sender()).balance.value
      == old(global<T>(sender()).balance.value) - amount;
}
```

Fig. 4. A simplified version of an example where verification led to an insight about a function. Without the `assert` marked "new," the specification fails to hold if `payee` and `sender` are the same, as explained in Sect. 6.

5 Specifications

The Move Prover has a basic specification language for individual functions. Specifications include classical Floyd-Hoare pre-conditions, post-conditions, and a new condition specifying when a function aborts. (We are expanding this functionality to include ghost variables and global invariants for modules.) These conditions are separated from the actual code, in "spec blocks," which are linked by name to the structure or function being specified, or to the containing module. Specifications never affect the execution of a module. A simplified example based on verifying a real Libra module appears in Fig. 4.

Pre-conditions and post-conditions are standard. Pre-conditions are introduced by the reserved word `requires` and post-conditions are introduced by `ensures`, and each is followed by a Boolean expression, in a syntax that is very similar to Move, which includes the usual relational and arithmetic operators, record field access, etc. A sub-expression after `ensures` can be enclosed in `old(...)`, causing the expression to be evaluated using the variable values in the program state immediately after entry to the function, instead of using the program state just before exit from the function. Move functions can return multiple values, so the expressions `return_1`, `return_2`, etc. represent those return values.

Formal verifiers for conventional programming languages treat run-time errors as bugs to be reported. However, as in most smart contract languages, performing an undefined operation in Move, such as division by zero, cancels the entire transaction with no effect on the state except the consumption of some currency to pay for the computational resources consumed by the code that was executed before the error occurred. In Libra, this event is called an *abort*. Aborts are not necessarily run-time errors in Move. They are the standard way to handle illegal transactions, such as trying to perform an operation that is not authorized by the sender of the transaction.

Instead of treating all possible abort conditions as bugs, the Move Prover allows the user to specify the conditions under which a function is expected to abort. This type of specification is introduced by the reserved word `aborts_if`, which is followed by the same kind of expressions that can appear after `requires`. When `aborts_if` P appears in the specification of a function, the Move Prover requires that the function aborts if and only if P holds. If multiple `aborts_if` conditions are specified, there is an error unless the function aborts if and only if the disjunction of all their conditions holds. (This current semantics of `aborts_if` is subject to change.)

There are two expressions that are specific to the Libra blockchain. The expression `exists<M::T>(A)` is true iff there is an instance of the type T from module M appearing under account A in the global state tree. In the example of Fig. 4, the first post-condition asserts that the payee account exists after a payment transaction (the payee account might not exist before the payment, in which case it is created). The expression `global<M::T>(A)` represents the value of type T from module M stored at account A. In the example, this construct accesses the balance values of the sender (the payer), to make sure that the balance covers the payment, and to assert that the payer account balance has decreased by the payment amount if the payment is successful.

Specification Translation. Specifications are translated into `requires` and `ensures` statements in Boogie and combined with the prelude (the Boogie model, see Sect. 4) and the translated Move bytecode for the program.

A global Boolean variable `$abort_flag` is introduced and assumed to be `false` at the beginning of each procedure. The Boogie code for each instruction sets this flag to `true` for conditions that cause abort, such as undefined operations or failures of explicit Move `assert` statements.

The specification translator combines, using logical disjunction, the conditions of all `aborts_if` statements into a single expression (called `condition` here), which is translated into the Boogie specifications `ensures condition ==> $abort_flag` and `ensures !condition ==> !$abort_flag`.

6 Evaluation

In this section, we report on our experience using the Move Prover. We first demonstrate that it can successfully be used on core modules in the Libra codebase.

Verifying Core Modules. We wrote specifications for all of the functions (25/25) in the Libra module and most of the functions (34/38) in the LibraAccount module (4 functions use features that are not yet supported: non-linear arithmetic and referencing data in the spec that does not appear in the code).[3] These are core modules of the Libra system, and their correct execution is crucial. The Move Prover was able to prove all of these specifications in under a minute, as shown below. The modules with their specifications are available in the Move Prover source tree.[4] The Libra and LibraAccount modules comprise nearly 1300 lines (including specifications). The total size of the generated Boogie files is a little over 14,000 lines, and the generated SMT files are around 52,000 lines. Writing these specifications was quite natural, thanks to the tree-based memory model and to the support for type-generics. Experiments were run on a machine with an Intel Core i9 processor with 8 cores @2.4 GHz and 32 GB RAM, running macOS Catalina.

Move Module	LoC	Boogie LoC	SMT LoC	Functions	Verified	Runtime
Libra	420	3875	11,688	25	25	2.99 s
LibraAccount	867	10,362	40293	38	34	46.66 s

Impact of Move Prover. The Move Prover is co-developed with the Move language itself (which is relatively stable) to ensure that contracts remain correct as the entire toolset evolves. The prover is used in continuous integration, and is beginning to be used to verify contracts in production. As of this writing, the Move Prover hasn't exposed any serious bugs. However, it has had an impact on how we understand code. An example is a function called `pay_from_sender` (a version with some specifications and comments omitted appears in Fig. 4). This function simply pays money from the account of the sender (who signed the transaction) to `payee`. In a previous version of the function, the Prover reported errors for two of the "obvious" specification properties shown. The first specification says that the function always aborts when paying zero Libra, because `deposit` aborts unless the amount is positive. However, in the earlier version, `create_account` handled the payment to deposit the amount in the account when the account did not yet exist, and that payment was allowed to be zero, violating the specification. The function was rewritten as it appears now, so that the same deposit code is called regardless of whether the payee account was newly created. The last specification says that the payer's account decreases by `amount` after a successful payment. This condition was violated when the payer and payee were the same, resulting in no decrease. Adding an assert (marked "new!" in the figure) to abort in that useless case makes the specification simpler.

[3] Two additional functions in LibraAccount are "native" which means that they are built-in and don't have any Move code. These are modeled directly in Boogie and are not included in the count here.

[4] To reproduce, run `cargo run -- -s . -- <libra|libra_account>.move` from `tests/sources/stdlib/modules` in the `move-prover` source tree.

7 Related Work

The only other formal verification framework for Move that we are aware of is described in [36], where a high-level approach and some case studies are described, but no implementation details are provided.

The closest work in the literature has been done in the context of verification of solidity smart contracts using Boogie. VERISOL [22] is one tool which formally verifies solidity smart contracts via a translation to Boogie. Its specification language is designed for the specific context of application policies, but general specifications can be given by using solidity assertions. SOLC-VERIFY [19,20] also uses Boogie to perform formal verification for solidity. It includes an annotation-based specification language and supports a larger feature-set of solidity than VERISOL. Interestingly, the formalization of the solidity persistent memory model presented in [20] is similar to our tree-based memory model for Move, though they were developed independently. One novelty of our model in comparison to theirs is its ability to handle generic functions as discussed in Sect. 4 (generics are supported in Move but not in solidity). Both VERISOL and SOLC-VERIFY target contracts written in solidity, and not in the Ethereum bytecode. In contrast, the Move Prover operates on the Move bytecode.

The solidity compiler itself includes a formal verification framework that works via a direct translation to SMT [2]. Several other tools have focused on specific vulnerability patterns, rather than user-defined specifications [16,28,34,40]. Other theoretical foundations have also been employed for the verification of solidity smart contracts. These include the $\mathbb{K}$ framework [35] (see, e.g., [21]), F* [29] (see, e.g., [11,18]), and proof assistants such as Coq [37] (see, e.g., [42,43]).

Formal verification of Rust [30] programs is also related to the Move Prover, as Move's type system has similar characteristics to Rust [30]. Prusti [4] is a tool that leverages Rust's type system information to verify Rust programs. It is based on a higher-level intermediate framework called Viper [33] (that internally uses Boogie in some scenarios). Other verification efforts for Rust employ a translation to LLVM and then leverage LLVM-based verification techniques (see, e.g., [6,27,39]).

8 Conclusion

In this paper, we introduced the Move Prover, a formal verification tool designed to be an integral part of the process of smart contract development for the Libra platform. Though our initial experience with the Move Prover is positive, there are many avenues for future work that we plan to pursue.

As Move continues to evolve, we expect that some constructs may be easier and more efficient to model by using custom SMT constructs. An example of this is the built-in vector type. Our current model requires the use of quantifiers to compare two vector objects. However, an SMT theory of sequences could be used to model vectors without needing to use quantifiers to define equality. We plan to investigate the use of richer (and possibly custom) SMT theories in our model.

The specifications we have written so far are *local* in the sense that they deal with only a single execution of a single Move function. However, some properties of the Libra blockchain are inherently *global* in nature, such as the fact that the total amount of currency should remain constant. We plan to investigate techniques for creating and checking such global specifications.

The current Prover is still in a prototype phase. But the goal is for it to be a product that is usable by everyone who is writing contracts for the Libra platform. We expect that there will be many challenges in producing a user-friendly, industrial-strength tool, but we also look forward to a future where formal specification and verification is a routine part of the development process for Move modules on the Libra blockchain.

References

1. Boogie. https://github.com/boogie-org/boogie
2. Alt, L., Reitwiessner, C.: SMT-based verification of solidity smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 376–388. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_28
3. Amsden, Z., et al.: The Libra Blockchain (2019). https://developers.libra.org/docs/the-libra-blockchain-paper
4. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. PACMPL 3(OOPSLA), 147:1–147:30 (2019)
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
6. Baranowski, M., He, S., Rakamarić, Z.: Verifying rust programs with SMACK. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 528–535. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_32
7. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
8. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 82–87. Association for Computing Machinery, New York (2005). https://doi.org/10.1145/1108792.1108813
9. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
11. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: PLAS@CCS, pp. 91–96. ACM (2016)
12. Blackshear, S., et al.: Move: A language with programmable resources (2019). https://developers.libra.org/docs/move-paper
13. Blackshear, S., et al.: Resources: A safe language abstraction for money (2020). https://arxiv.org/abs/2004.05106

14. Buterin, V.: Critical update re DAO (2016). https://ethereum.github.io/blog/2016/06/17/critical-update-re-dao-vulnerability
15. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on Ethereum systems security: vulnerabilities, attacks and defenses. CoRR abs/1908.04507 (2019)
16. ConsenSys: Mythril Classic: Security analysis tool for Ethereum smart contracts. https://github.com/skylightcyber/mythril-classic
17. Girard, J.: Linear logic. Theor. Comput. Sci. **50**(1), 1–101 (1987)
18. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
19. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. CoRR abs/1907.04262 (2019)
20. Hajdu, Á., Jovanović, D.: SMT-friendly formalization of the solidity memory model. ESOP 2020. LNCS, vol. 12075, pp. 224–250. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_9
21. Hildenbrandt, E., et al.: KEVM: a complete formal semantics of the Ethereum virtual machine. In: CSF, pp. 204–217. IEEE Computer Society (2018)
22. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR abs/1812.08829 (2018)
23. Leino, K.R.M.: This is boogie 2 (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/, manuscript KRML 178
24. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_26
25. Libra. https://github.com/libra/libra
26. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1997)
27. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of rust programs by symbolic execution. In: INDIN, pp. 108–114. IEEE (2018)
28. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
29. Maillard, K., et al.: Dijkstra monads for all. In: 24th ACM SIGPLAN International Conference on Functional Programming (ICFP) (2019). https://arxiv.org/abs/1903.01237
30. Matsakis, N.D., Klock II, F.S.: The rust language. Ada Lett. **34**(3), 103–104 (2014). https://doi.org/10.1145/2692956.2663188
31. Meijer, E., Wa, R., Gough, J.: Technical overview of the common language runtime (2000)
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017)
34. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC, pp. 653–663. ACM (2018)

35. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
36. Synthetic Minds Blog: Verifying smart contracts in the move language (2019). https://synthetic-minds.com/pages/blog/blog-2019-09-11.html
37. The Coq development team: The coq proof assistant reference manual version 8.9 (2019). https://coq.inria.fr/distrib/current/refman/
38. The Libra Association: An Introduction to Libra (2019). https://libra.org/en-us/whitepaper
39. Toman, J., Pernsteiner, S., Torlak, E.: Crust: a bounded verifier for rust (N). In: ASE, pp. 75–80. IEEE Computer Society (2015)
40. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: ACM Conference on Computer and Communications Security, pp. 67–82. ACM (2018)
41. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014). https://ethereum.github.io/yellowpaper/paper.pdf
42. Yang, Z., Lei, H.: Formal process virtual machine for smart contracts verification. CoRR abs/1805.00808 (2018)
43. Yang, Z., Lei, H.: Fether: an extensible definitional interpreter for smart-contract verifications in Coq. IEEE Access **7**, 37770–37791 (2019)

End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract

Daejun Park[1(✉)], Yi Zhang[1,2], and Grigore Rosu[1,2]

[1] Runtime Verification, Inc., Urbana, IL, USA
`daejun.park@runtimeverification.com`
[2] University of Illinois at Urbana-Champaign, Urbana, IL, USA
`{yzhng173,grosu}@illinois.edu`

Abstract. We report our experience in the formal verification of the deposit smart contract, whose correctness is critical for the security of Ethereum 2.0, a new Proof-of-Stake protocol for the Ethereum blockchain. The deposit contract implements an incremental Merkle tree algorithm whose correctness is highly nontrivial, and had not been proved before. We have verified the correctness of the compiled bytecode of the deposit contract to avoid the need to trust the underlying compiler. We found several critical issues of the deposit contract during the verification process, some of which were due to subtle hidden bugs of the compiler.

1 Introduction

The deposit smart contract [14] is a gateway to join Ethereum 2.0 [15] that is a new sharded Proof-of-Stake (PoS) protocol which at its early stage, lives in parallel with the existing Proof-of-Work (PoW) chain, called Ethereum 1.x chain. Validators drive the entire PoS chain, called Beacon chain, of Ethereum 2.0. To be a validator, one needs to deposit a certain amount of Ether, as a "stake", by sending a transaction (over the Ethereum 1.x network) to the deposit contract. The deposit contract records the history of deposits, and locks all the deposits in the Ethereum 1.x chain, which can be later claimed at the Beacon chain of Ethereum 2.0.[1] Note that the deposit contract is a one-way function; one can move her funds from Ethereum 1.x to Ethereum 2.0, but not vice versa.

The deposit contract, written in Vyper [19], employs the Merkle tree [30] data structure to efficiently store the deposit history, where the tree is *dynamically* updated (i.e., leaf nodes are incrementally added in order from left to right) whenever a new deposit is received. The Merkle tree employed in this contract is very large: it has height 32, so it can store up to 2^{32} deposits. Since the size of the Merkle tree is huge, it is not practical to reconstruct the whole tree every time a new deposit is received.

To reduce both time and space complexity, thus saving the gas[2] cost significantly, the contract implements an *incremental Merkle tree algorithm* [6]. The

[1] This deposit process will change at a later stage.
[2] In Ethereum, gas refers to the fee to execute a transaction or a smart contract on the blockchain. The amount of gas fee depends on the size of the payloads.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 151–164, 2020.
https://doi.org/10.1007/978-3-030-53288-8_8

incremental algorithm enjoys $O(h)$ time and space complexity to reconstruct (more precisely, compute the root of) a Merkle tree of height h, while a naive algorithm would require $O(2^h)$ time or space complexity. The efficient incremental algorithm, however, leads to the deposit contract implementation being unintuitive, and makes it non-trivial to ensure its correctness. The correctness of the deposit contract, however, is critical for the security of Ethereum 2.0, since it is a gateway for becoming a validator. Considering the utmost importance of the deposit contract for the Ethereum blockchain, formal verification is demanded to ultimately guarantee its correctness.

In this paper, we present our formal verification of the deposit contract.[3] The scope of verification is to ensure the correctness of the contract bytecode within a single transaction, without considering transaction-level or off-chain behaviors. We take the compiled bytecode as the verification target to avoid the need to trust the compiler.[4]

We adopt a refinement-based verification approach. Specifically, our verification effort consists of the following two tasks:

– Verify that the incremental Merkle tree algorithm implemented in the deposit contract is *correct* w.r.t. the original full-construction algorithm.
– Verify that the compiled bytecode is *correctly generated* from the source code of the deposit contract.

Intuitively, the first task amounts to ensuring the correctness of the contract source code, while the second task amounts to ensuring the compiled bytecode being a sound refinement of the source code (i.e., translation validation of the compiler). This refinement-based approach allows us to avoid reasoning about the complex algorithmic details, especially specifying and verifying loop invariants, directly at the bytecode level. This separation of concerns helped us to save a significant amount of verification effort. See Sect. 1.1 for more details.

Challenges. Formally verifying the deposit contract was challenging. First, the algorithm employed in the contract is sophisticated and its correctness is not straightforward to prove. Indeed, we found a critical bug in the algorithm implementation which had been not detected by existing tests (Sect. 3.1).

Second, we had to take the compiled bytecode as the verification target, which is much larger (consisting of ∼3,000 instructions) and more complex than the source code. The source-code-level verification was not accepted by the customer for the end-to-end correctness guarantee, especially considering the fact that the compiler is not mature enough [11]. Indeed, we found several new critical bugs in the compiler during the formal verification process (Sect. 3.2).

Third, we had to consider not only the functional correctness, but also security properties of the contract. That is, we had to identify the behaviors of the contract in exceptional cases, and check if they are exploitable. We found a bug of the contract in case that it receives invalid inputs (Sect. 3.3).

[3] This was done as part of a contract funded by the Ethereum Foundation [16].

[4] Indeed, we found several new critical bugs [41–44] of the Vyper compiler in the process of formal verification. See Sect. 3 for more details.

Finally, we had to take into account potential future changes in the Ethereum blockchain system (called hard-forks). That is, we had to verify that the compiled bytecode will work not only in the current system, but also in any future version of the system that employs a different gas fee schedule. Considering such potential changes of the system required us to generalize the semantics of bytecode execution. We also found a bug regarding that (Sect. 3.4).

1.1 Our Refinement-Based Verification Approach

We illustrate our refinement-based formal verification approach used in the deposit contract verification. We present our approach using the K framework and its verification infrastructure [46,52,55], but it can be applied to other program verification frameworks.

Let us consider a `sum` program that computes the summation from 1 to n:

```
int sum(int n) { int s = 0; int i = 1;
                 while(i <= n) { s = s + i; i = i + 1; } return s; }
```

Given this program, we first manually write an abstract model of the program in the K framework [52]. Such a K model is essentially a state transition system of the program, and can be written as follows:

$$\begin{aligned}
&\texttt{rule: sum}(n) \Rightarrow \texttt{loop(s: } 0\texttt{, i: } 1\texttt{, n: } n\texttt{)}\\
&\texttt{rule: loop(s: } s\texttt{, i: } i\texttt{, n: } n\texttt{)} \Rightarrow \texttt{loop(s: } s+i\texttt{, i: } i+1\texttt{, n: } n\texttt{) when } i \leq n\\
&\texttt{rule: loop(s: } s\texttt{, i: } i\texttt{, n: } n\texttt{)} \Rightarrow \texttt{return(}s\texttt{) when } i > n
\end{aligned}$$

These transition rules correspond to the initialization, the `while` loop, and the return statement, respectively. The indexed tuple $(\texttt{s}: s, \texttt{i}: i, \texttt{n}: n)$ represents the state of the program variables `s`, `i`, and `n`.[5]

Then, given the abstract model, we specify the functional correctness property in reachability logic [54], as follows:

$$\texttt{claim: sum}(n) \Rightarrow \texttt{return(}\tfrac{n(n+1)}{2}\texttt{) when } n > 0$$

This reachability claim says that `sum`(n) will eventually return $\frac{n(n+1)}{2}$ in all possible execution paths, if n is positive. We verify this specification using the K reachability logic theorem prover [55], which requires us only to provide the following loop invariant:[6]

$$\texttt{invariant: loop(s: } \tfrac{i(i-1)}{2}\texttt{, i: } i\texttt{, n: } n\texttt{)} \Rightarrow \texttt{return(}\tfrac{n(n+1)}{2}\texttt{) when } 0 < i \leq n+1$$

Once we prove the desired property of the abstract model, we manually refine the model to a bytecode specification, by translating each transition rule of the abstract model into a reachability claim at the bytecode level, as follows:

[5] Note that this abstract model can be also automatically derived by instantiating the language semantics with the particular program, if a formal semantics of the language is available (in the K framework).

[6] The loop invariants in reachability logic mentioned here look different from those in Hoare logic. See the comparison between the two logic proof systems in [55, Section 4]. These loop invariants can be also seen as transition invariants [48].

$$\begin{array}{l}
\texttt{claim: evm(pc: } \mathsf{pc}_{\mathsf{begin}}\texttt{, calldata: \#bytes(32, } n\texttt{), stack: [], } \cdots\texttt{)} \\
\quad\quad \Rightarrow \texttt{evm(pc: } \mathsf{pc}_{\mathsf{loophead}}\texttt{, stack: [0, 1, } n\texttt{], } \cdots\texttt{)} \\
\texttt{claim: evm(pc: } \mathsf{pc}_{\mathsf{loophead}}\texttt{, stack: [} s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)} \\
\quad\quad \Rightarrow \texttt{evm(pc: } \mathsf{pc}_{\mathsf{loophead}}\texttt{, stack: [} s+i\texttt{, } i+1\texttt{, } n\texttt{], } \cdots\texttt{) when } i \le n \\
\texttt{claim: evm(pc: } \mathsf{pc}_{\mathsf{loophead}}\texttt{, stack: [} s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)} \\
\quad\quad \Rightarrow \texttt{evm(pc: } \mathsf{pc}_{\mathsf{end}}\texttt{, stack: [], output: \#bytes(32, } s\texttt{), } \cdots\texttt{) when } i > n
\end{array}$$

Here, the indexed tuple `evm(pc:_, calldata:_, stack:_, output:_)` represents (part of) the Ethereum Virtual Machine (EVM) state, and `#bytes(`N, V`)` denotes a sequence of N bytes of the two's complement representation of V.

We verify this bytecode specification against the compiled bytecode using the same K reachability theorem prover [46,55]. Note that no loop invariant is needed in this bytecode verification, since each reachability claim involves only a bounded number of execution steps—specifically, the second claim involves only a single iteration of the loop.

Then, we manually prove the soundness of the refinement, which can be stated as follows: *for any EVM states* σ_1 *and* σ_2*, if* $\sigma_1 \Rightarrow \sigma_2$*, then* $\alpha(\sigma_1) \Rightarrow \alpha(\sigma_2)$, where the abstraction function α is defined as follows:

$$\begin{array}{l}
\alpha(\texttt{evm(pc: } \mathsf{pc}_{\mathsf{begin}}\texttt{, calldata: \#bytes(32, } n\texttt{), stack: [], } \cdots\texttt{)}) = \texttt{sum(}n\texttt{)} \\
\alpha(\texttt{evm(pc: } \mathsf{pc}_{\mathsf{loophead}}\texttt{, stack: [} s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)}) = \texttt{loop(s: } s\texttt{, i: } i\texttt{, n: } n\texttt{)} \\
\alpha(\texttt{evm(pc: } \mathsf{pc}_{\mathsf{end}}\texttt{, stack: [], output: \#bytes(32, } s\texttt{), } \cdots\texttt{)}) = \texttt{return(}s\texttt{)}
\end{array}$$

Putting all the results together, we finally conclude that the compiled bytecode will return `#bytes(32,`$\frac{n(n+1)}{2}$`)`.

Note that the abstract model and the compiler are *not* in the trust base, thanks to the refinement, while the K reachability logic theorem prover [46,55] and the formal semantics of EVM [24] are.

2 Formal Verification of the Deposit Contract

Following the refinement-based approach illustrated in Sect. 1.1, we first formalized the main business logic of the deposit contract (i.e., the incremental Merkle tree algorithm), and proved its correctness. Then we refined the formal model into a bytecode specification, and verified the compiled bytecode of the deposit contract against the refined specification. From these, we concluded the correctness of the deposit contract bytecode.

2.1 Incremental Merkle Tree Algorithm

We briefly describe the incremental Merkle tree algorithm of the deposit contract. Due to space limitations, we omit the formalization of the algorithm and the formal proof of the correctness, and refer the readers to our companion technical report [45] for the full details.

A Merkle tree [30] is a perfect binary tree [34] where leaf nodes store the hash of data, and non-leaf nodes store the hash of their children. A *partial Merkle tree*

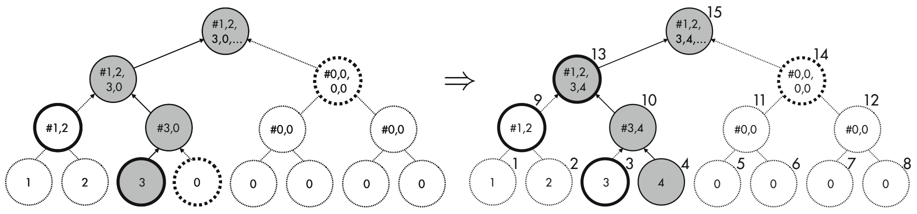


Fig. 1. Illustration of the incremental Merkle tree algorithm. Node numbers are labeled in the upper-right corner of each node.

up-to m is a Merkle tree whose first (leftmost) m leaves are filled with data hashes and the other leaves are empty and filled with zeros. The incremental Merkle tree algorithm takes as input a partial Merkle tree up-to m and a new data hash, and inserts the new data hash into the $(m+1)^{\text{th}}$ leaf, resulting in a partial Merkle tree up-to $m+1$.

Figure 1 illustrates the algorithm, showing how the given partial Merkle tree up-to 3 (shown in the left) is updated to the resulting partial Merkle tree up-to 4 (in the right) when a new data hash is inserted into the 4^{th} leaf node. The key idea of the algorithm is that only the path from the new leaf to the root (i.e., the gray nodes) needs to be computed (hence linear-time), and moreover the path can be computed by using only the left (i.e., node 3 and node 9) or right (i.e., node 14) sibling of each node in the path, which are only nodes that the algorithm maintains (hence linear-space). Refer to [45] for the full details.

2.2 Bytecode Verification of the Deposit Contract

Now we present the formal verification of the compiled bytecode of the deposit contract. The bytecode verification ensures that the compiled bytecode is a sound refinement of the source code. This rules out the need to trust the compiler.

As illustrated in Sect. 1.1, we first manually refined the abstract model (in which we proved the algorithm correctness) to the bytecode specification. For the refinement, we consulted the ABI interface standard [13] (to identify, e.g., `calldata` and `output` in the illustrating example of Sect. 1.1), as well as the bytecode (to identify, e.g., the `pc` and `stack` information).[7] Then, we used the KEVM verifier [46] to verify the compiled bytecode against the refined specification. We adopted the KEVM verifier to reason about all possible corner-case behaviors of the compiled bytecode, especially those introduced by certain unintuitive and questionable aspects of the underlying Ethereum Virtual Machine (EVM) [60]. This was possible because the KEVM verifier is derived from a complete formal semantics of the EVM, called KEVM [24]. Our formal specification and verification artifacts are publicly available at [50].

[7] However, we want to note that the Vyper compiler can be augmented to extract such information, which can automate the refinement process to a certain extent. We leave that as future work.

Let us elaborate on specific low-level behaviors verified against the bytecode. In addition to executing the incremental Merkle tree algorithm, most of the functions perform certain additional low-level tasks, and we verified that such tasks are correctly performed. Specifically, for example, given deposit data,[8] the `deposit` function computes its 32-byte hash (called Merkleization) according to the SimpleSerialize (SSZ) specification [18]. The leaves of the Merkle tree store only the computed hashes instead of the original deposit data. The `deposit` function also emits a `DepositEvent` log that contains the original deposit data, where the log message needs to be encoded as a byte sequence following the contract event ABI specification [13]. Other low-level operations performed by those functions that we verified include: correct zero-padding for the 32-byte alignment, correct conversions from big-endian to little-endian, input bytes of the SHA2-256 hash function being correctly constructed, and return values being correctly serialized to byte sequences according to the ABI specification [13].

We also verified a liveness property that the contract is always able to accept a new (valid) deposit as long as a sufficient amount of gas is provided. This liveness is not trivial since it needs to hold even in any future hard-fork where the gas fee schedule is changed. Indeed, we found a bug that violates the liveness. See Sect. 3.4 for more details.

Our formal specification includes both positive and negative behaviors. The positive behaviors describe the desired behaviors of the contracts in a legitimate input state. The negative behaviors, on the other hand, describe how the contracts handle exceptional cases (e.g., when benign users feed invalid inputs by mistake, or malicious users feed crafted inputs to take advantage of the contracts). The negative behaviors are mostly related to security properties.

For the full specification of the verified bytecode behaviors, refer to [49].

3 Findings and Lessons Learned

In the course of our formal verification effort, we found subtle bugs [35–37] of the deposit contract, as well as a couple of refactoring suggestions [38–40] that can improve the code readability and reduce the gas cost. The subtle bugs of the deposit contract are partly due to bugs of the Vyper compiler [41–44] that we newly found (and reported to the Vyper team) in the verification process.

Below we elaborate on the bugs we found and lessons we learned along the way. We note that all the bugs of the deposit contract have been reported, confirmed, and properly fixed in the latest version (v0.11.2).

3.1 Maximum Number of Deposits

In the original version of the contract that we were asked to verify, a bug is triggered when all of the leaf nodes of a Merkle tree are filled with deposit

[8] Each deposit data consists of the public key, the withdrawal credentials, the deposit amount, and the signature of the deposit owner.

data, in which case the contract (specifically, the `get_deposit_root` function) incorrectly computes the root hash of a tree, returning the zero root hash (i.e., the root hash of an empty Merkle tree) regardless of the content of leaf nodes. For example, suppose that we have a Merkle tree of height 2, which has four leaf nodes, and every leaf node is filled with certain deposit data, say v_1, v_2, v_3, and v_4, respectively. Then, while the correct root hash of the tree is $\mathsf{hash}(\mathsf{hash}(v_1, v_2), \mathsf{hash}(v_3, v_4))$, the `get_deposit_root` function returns $\mathsf{hash}(\mathsf{hash}(0, 0), \mathsf{hash}(0, 0))$, which is incorrect.

Due to the complex logic of the code, it is non-trivial to properly fix this bug without significantly rewriting the code, and thus we suggested a workaround that simply forces to never fill the last leaf node, i.e., accepting only $2^h - 1$ deposits at most, where h is the height of a tree. We note that, however, it is infeasible in practice to trigger this buggy behavior in the current setting, since the minimum deposit amount is 1 Ether and the total supply of Ether is less than 130M which is much smaller than 2^{32}, thus it is not feasible to fill all the leaves of a tree of height 32. Nevertheless, this bug has been fixed by the contract developers as we suggested, since the contract may be used in other settings in which the buggy behavior can be triggered and an exploit may be possible. Refer to [37] for more details.

We also want to note that this bug was quite subtle to catch. Indeed, we had initially thought that the original code was correct until we failed to write a formal proof of the correctness theorem. The failure of our initial attempt to prove the correctness led us to identify a missing premise that was needed for the theorem to hold, from which we could find the buggy behavior scenario, and suggested the bugfix. This experience reconfirms the importance of formal verification. Although we were not "lucky" to find this bug when we had eyeball-reviewed the code, which is all traditional security auditors do, the formal verification process thoroughly guided and even "forced" us to find it eventually.

3.2 ABI Standard Conformance of `get_deposit_count` Function

In the previous version, the `get_deposit_count` function does not conform to the ABI standard [13], where its return value contains incorrect zero-padding [35], due to a Vyper compiler bug [41]. Specifically, in the buggy version of the compiled bytecode, the `get_deposit_count` function, whose return type is `bytes[8]`, returns a byte sequence of length 96, where the last byte is `0x20` while it should be `0x00`. According to the ABI specification [13], the last 24 bytes must be all zero, serving as zero-pad for the 32-byte alignment. Thus the return value does not conform to the ABI standard. This is problematic because any contract (written in either Solidity or Vyper) that calls to (the buggy version of) the deposit contract, expecting that the `deposit_count` function conforms to the ABI standard, could have misbehaved.[9]

[9] The returned byte sequence, including the incorrect last byte, is copied to the caller's memory. If the caller reuses the last byte assuming that it is zero, the garbage value will be passed around, which may break the business logic of the caller.

This buggy behavior is mainly due to a subtle Vyper compiler bug [41] that fails to correctly compile a function whose return type is `bytes[`n`]` where $n < 16$. This leads to the compiled function returning a byte sequence with insufficient zero-padding as mentioned above, failing to conform to the ABI standard.

We note that this bug could not have been detected if we did not take the bytecode as the verification target. This reconfirms that the bytecode-level verification is critical to ensure the ultimate correctness (unless we formally verify the underlying compiler), because we cannot (and should not) trust the compiler.

3.3 Checking Well-Formedness of Calldata

The calldata decoding process in the previous version of the compiled bytecode does not have sufficient runtime-checks for the well-formedness of calldata. As such, it fails to detect certain ill-formed calldata, causing invalid deposit data to be put into the Merkle tree. This is problematic especially when clients make mistakes and send deposit transactions with incorrectly encoded calldata, which may result in losing their deposit fund.

Specifically, we found a counter-example ill-formed calldata whose size (196 bytes) is much less than that of well-formed calldata (356 bytes). The problem, however, is that the `deposit` function does *not* reject the ill-formed calldata, but simply inserts certain invalid (garbage) deposit data in the Merkle tree. Since the invalid deposit data cannot pass the signature validation later, no one can claim the deposited fund associated with this, and the deposit owner loses the fund. Note that this happens even though the `deposit` function employs assertions at the beginning of the function that ensures the size of each of the arguments is correct, which turned out to not work as expected.

This problem would not exist if the Vyper compiler thoroughly generated runtime checks to ensure the well-formedness of calldata.[10] However, since it was not trivial to fix the compiler to generate such runtime checks, we suggested several ways to improve the deposit contract source code to prevent this behavior without fixing the compiler. After careful discussion with the deposit contract development team, we together decided to employ a checksum-based approach where the `deposit` function takes as an additional input a checksum for the deposit data, and rejects any ill-formed calldata using the checksum. The checksum-based approach is the least intrusive and the most gas-efficient of all the suggested fixes. For more details of other suggested fixes, refer to [36].

We note that this issue was found when we were verifying the negative behaviors of the deposit contract. This shows the importance of having the formal specification to include not only positive but also negative behaviors.

[10] The compiler developers failed to consider the case when the given calldata is not correctly encoded. For example, while the header of calldata contains offsets (i.e., pointers) to the positions of data elements, it could be the case that certain offsets are beyond the calldata range. In that case, the calldata can be accessed outside its bounds, due to the missing runtime-checks.

3.4 Liveness

As mentioned in Sect. 2.2, the previous version of the deposit contract fails to satisfy a liveness property in that it may not be able to accept a new deposit, even if it is valid, in a certain future hard-fork that updates the gas fee schedule. This was mainly due to another subtle Vyper compiler bug [44] that generates bytecode where a hard-coded amount of gas is supplied when calling to certain precompiled contracts. Although this hard-coded amount of gas is sufficient in the current hard-fork (code-named Istanbul [17]), it may not be sufficient in a certain future hard-fork that increases the gas fee schedule of the precompiled contracts. In such a future hard-fork, the previous version of the deposit contract will always fail due to the out-of-gas exception, regardless of how much gas is initially supplied. Refer to [44] for more details.

We admit that we could not find this issue until the deposit contract development team carefully reviewed and discussed with us the formal specification [49] of the bytecode. Initially, we considered only the behaviors of the bytecode in the current hard-fork, without identifying the requirement that the contract bytecode should work in any future hard-fork. We identified the missing requirement, and found this liveness issue, at a very late stage of the formal verification process, which delayed the completion of formal verification.

This experience essentially illustrates the well-known problem caused by the gap between the intended behaviors (that typically exists only informally) by developers, and the formal specification written by verification engineers. To reduce this gap, the two groups should work closely together, or ideally, developers should write their own specifications in the first place. For the former, the formal verification process should involve developers more frequently. For the latter, the formal verification tools should become much easier to use without requiring advanced knowledge of formal methods. We leave both as future work.

3.5 Discussion

Verification Effort. The net effort for formal verification took 7 person-weeks (excluding various discussions with developers, reporting bugs and following-up, especially for compiler bugs, etc.), where the algorithm correctness proof took 2 person-weeks, and the bytecode verification took 5 person-weeks. This includes the time spent on writing specifications as well. The bytecode specification consists of $\sim$1,000 LOC (excluding comments), in addition to auxiliary lemmas consisting of $\sim$200 LOC. The size of the source code is $\sim$100 LOC, and the number of instructions in the compiled bytecode is $\sim$3,000.

Trust Base. The validity of the bytecode verification result assumes the correctness of the bytecode specification and the KEVM verifier. The algorithm correctness proof is partially mechanized—only the proof of major lemmas are mechanized in the K framework. The non-mechanized proofs are included in our trust base. The Vyper compiler is *not* in the trust base.

Continuous Verification. The verification target contract was a moving target. Even if the contract code had been frozen before starting the formal verification process, the code (both source code or bytecode) was updated in the middle of the verification process, to fix bugs found during the process. Indeed, we found several bugs in both the contract and the compiler, and each time we found a bug, we had to re-verify the newly compiled bytecode that fixes the bug. Here the problem was the overhead of re-verification. About 20% of the bytecode verification effort was spent on re-verification.

The re-verification overhead could have been reduced by automatically adjusting formal specifications to updated bytecode, and/or making specifications as independent of the specific details of the bytecode as possible. For example, the current bytecode specification employs specific program-counter (PC) values to refer to some specific positions of the bytecode, especially when specifying loop invariants. Most of such PC values need to be updated whenever the bytecode is modified. The re-verification overhead could have been reduced by automatically updating such PC values, or even having the specification refer to specific positions without using PC values. We leave this as future work.

4 Related Work

Static Analysis and Verification of Smart Contracts. There have been proposed many static analysis tools [5,10,20,25,28,29,32,57,58] that are designed to automatically detect a certain fixed set of bugs and vulnerabilities of smart contracts, at the cost of generality and expressiveness. VerX [47] can verify past-time linear temporal properties over multiple runs of smart contracts, but it requires the target contracts to be effectively loop-free.

There also have been proposed verification tools that allow us to specify and verify arbitrary functional correctness and/or security properties, such as [3,22] based on the F* proof assistant [1,56] based on Isabelle/HOL [33], the KEVM verifier [46] based on the K framework [52], and VeriSol [27] based on Boogie [2]. The KEVM verifier has also been used to verify high-profile and challenging smart contracts [51], including a multi-signature wallet called Gnosis Safe [21], a decentralized token exchange called Uniswap [59], and a partial consensus mechanism called Casper FFG [7].

Verification of Systems Software. There are many success stories of formal verification of systems software, from OS kernels [23,26,31], to file systems [8,53], to cryptographic code [4]. While most of the verified systems code is either synthesized from specifications, or implemented (or adjusted) to be verification-friendly, there also exist efforts [9,12] to verify actual production code as is. Such efforts are necessary especially when the production code is highly performance-critical and/or existing development processes are hard to change to help produce verification-friendly code. The deposit contract we verified was given to us at the code-frozen stage, and also performance-critical (especially in terms of the gas cost), and thus we took and verified the given production-ready code as is, without any modification except for fixing bugs.

References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/hol. In: Proceedings of the 7th ACM International Conference on Certified Programs and Proofs, CPP 2018 (2018)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: 4th International Symposium on Formal Methods for Components and Objects, FMCO 2005, Amsterdam, The Netherlands, November 1–4, 2005, Revised Lectures (2005)
3. Bhargavan, K., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS 2016 (2016)
4. Bond, B., et al.: Vale: verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017 (2017)
5. Brent, L., et al.: Vandal: a scalable security analysis framework for smart contracts. CoRR abs/1809.03981 (2018)
6. Buterin, V.: Progressive Merkle Tree. https://github.com/ethereum/research/blob/master/beacon_chain_impl/progressive_merkle_tree.py
7. Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR abs/1710.09437 (2017)
8. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the FSCQ file system. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4–7, 2015 (2015)
9. Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26
10. ConsenSys Diligence: MythX. https://mythx.io/
11. ConsenSys Diligence: Vyper Security Review. https://diligence.consensys.net/audits/2019/10/vyper/
12. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 467–486. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_28
13. Ethereum Foundation: Contract ABI Specification. https://solidity.readthedocs.io/en/v0.6.1/abi-spec.html
14. Ethereum Foundation: Ethereum 2.0 Deposit Contract. https://github.com/ethereum/eth2.0-specs/blob/v0.11.2/deposit_contract/contracts/validator_registration.vy
15. Ethereum Foundation: Ethereum 2.0 Specifications. https://github.com/ethereum/eth2.0-specs
16. Ethereum Foundation: Ethereum Foundation Spring 2019 Update. https://blog.ethereum.org/2019/05/21/ethereum-foundation-spring-2019-update/
17. Ethereum Foundation: Hardfork Meta: Istanbul. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md
18. Ethereum Foundation: SimpleSerialize (SSZ). https://github.com/ethereum/eth2.0-specs/tree/dev/ssz
19. Ethereum Foundation: Vyper. https://vyper.readthedocs.io

20. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019 (2019)
21. Gnosis Ltd.: Gnosis Safe. https://safe.gnosis.io/
22. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Proceedings of the 7th International Conference on Principles of Security and Trust, POST 2018 (2018)
23. Gu, R., et al.: Certikos: an extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2–4, 2016 (2016)
24. Hildenbrandt, E., et al.: KEVM: a complete semantics of the Ethereum virtual machine. In: Proceedings of the 31st IEEE Computer Security Foundations Symposium, CSF 2018 (2018)
25. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018 (2018)
26. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11–14, 2009 (2009)
27. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR abs/1812.08829 (2018)
28. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016 (2016)
29. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 450–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_33
30. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
31. Nelson, L., et al.: Hyperkernel: push-button verification of an OS kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28–31, 2017 (2017)
32. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03–07, 2018 (2018)
33. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL- A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
34. NIST: Perfect Binary Tree. https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html
35. Park, D.: Ethereum 2.0 deposit contract issue 1341: non ABI-standard return value of get_deposit_count of deposit contract. https://github.com/ethereum/eth2.0-specs/issues/1341
36. Park, D.: Ethereum 2.0 deposit contract issue 1357: Ill-formed call data to deposit contract can add invalid deposit data. https://github.com/ethereum/eth2.0-specs/issues/1357

37. Park, D.: Ethereum 2.0 deposit contract issue 26: maximum deposit count. https://github.com/ethereum/deposit_contract/issues/26
38. Park, D.: Ethereum 2.0 deposit contract issue 27: redundant assignment in init(). https://github.com/ethereum/deposit_contract/issues/27
39. Park, D.: Ethereum 2.0 deposit contract issue 28: loop fusion optimization. https://github.com/ethereum/deposit_contract/issues/28
40. Park, D.: Ethereum 2.0 deposit contract issue 38: a refactoring suggestion for the loop of deposit(). https://github.com/ethereum/deposit_contract/issues/38
41. Park, D.: Vyper Issue 1563: Insufficient zero-padding bug for functions returning byte arrays of size < 16. https://github.com/vyperlang/vyper/issues/1563
42. Park, D.: Vyper Issue 1599: Off-by-one error in zero_pad(). https://github.com/vyperlang/vyper/issues/1599
43. Park, D.: Vyper Issue 1610: Non-semantics-preserving refactoring for zero_pad(). https://github.com/vyperlang/vyper/issues/1610
44. Park, D.: Vyper Issue 1761: Potentially insufficient gas stipend for precompiled contract calls. https://github.com/vyperlang/vyper/issues/1761
45. Park, D., Zhang, Y., Rosu, G.: End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract. http://hdl.handle.net/2142/107129
46. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for Ethereum VM Bytecode. In: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018 (2018)
47. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety Verification of Smart Contracts. https://files.sri.inf.ethz.ch/website/papers/sp20-verx.pdf
48. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, LICS 2004 (2004)
49. Runtime Verification Inc.: Bytecode Behavior Specification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/verified-smart-contracts/blob/master/deposit/bytecode-verification/deposit-spec.ini.md
50. Runtime Verification Inc.: Formal Verification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/verified-smart-contracts/tree/master/deposit
51. Runtime Verification Inc.: Formally Verified Smart Contracts. https://github.com/runtimeverification/verified-smart-contracts
52. Serbanuta, T., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K primer (version 3.3). Electr. Notes Theor. Comput. Sci. **304**, 57–80 (2014)
53. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2–4, 2016 (2016)
54. Stefanescu, A., Ciobaca, S., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-Path Reachability Logic. Logical Methods in Computer Science **15**(2), (2019)
55. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016 (2016)
56. Swamy, N., et al.: Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016 (2016)

57. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: static analysis of Ethereum smart contracts. In: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27–June 3, 2018 (2018)
58. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018 (2018)
59. Uniswap: Uniswap Exchange Protocol. https://uniswap.io/
60. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf

Stratified Abstraction of Access Control Policies

John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek(✉), Ranjit Jhala, Kasper Luckow, Sean McLaughlin, Madhav Menon, Daniel Peebles, Ujjwal Pugalia, Neha Rungta, Cole Schlesinger, Adam Schodde, Anvesh Tanuku, Carsten Varming, and Deepa Viswanathan

Amazon Web Services, Seattle, USA
`gacek@amazon.com`

Abstract. The shift to cloud-based APIs has made application security critically depend on understanding and reasoning about *policies* that regulate access to cloud resources. We present *stratified predicate abstraction*, a new approach that summarizes complex security policies into a compact set of positive and declarative statements that precisely state *who* has access to a resource. We have implemented stratified abstraction and deployed it as the engine powering AWS's IAM Access Analyzer service, and hence, demonstrate how formal methods and SMT can be used for security policy *explanation*.

1 Introduction

A growing number of developers are using cloud-based implementations of basic resources like associative arrays, encryption, storage, queuing, and event-driven execution, to engineer client applications. For example, millions of Amazon Web Services (AWS) customers use cloud APIs like Amazon SQS for queues, Amazon S3 for storage, AWS KMS for crypto key management, Amazon DynamoDB for associative arrays, and AWS Lambda for executing functions in a pure virtualized environment. This shift to the cloud has made application security critically depend upon deeply understanding and reasoning about *policies* that regulate how different principals are allowed to access cloud resources. AWS users, for example, configure principals in the Identity and Access Management (IAM) service. The users define which requests are allowed access via *resource policies* which allow some resources to be purposefully shared with the entire internet, while restricting access to others to limited sets of identities.

The IAM policy language has many features that are essential to allow users to build a wide array of possible applications. Some of these features make reasoning about policies challenging. First, individual policy elements can use regular expressions, negation, and conditionals. Second, the policy elements can interact with each other in subtle ways that make the net effect of a policy unclear. Previously, we developed ZELKOVA [2], a tool that encodes policies as logical

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 165–176, 2020.
https://doi.org/10.1007/978-3-030-53288-8_9

formulas and then uses SMT solvers [3,8] to answer questions about policies, *e.g.* whether a particular policy is *correct*, too strict, or too permissive. While Zelkova can be queried to *explore* the properties of policies *e.g.* whether some resource is "publicly" accessible, our experience shows that formal policy analysis remains challenging as users must have sufficient technical sophistication to realize the criteria important to them *and* be able to formalize the above as Zelkova queries.

```
- Effect: Allow
  Condition:
    StringEquals:
      SrcVpc:
        - vpc-a
        - vpc-b
- Effect: Allow
  Condition:
    StringEquals:
      OrgID: o-2
- Effect: Deny
  Condition:
    StringEquals:
      SrcVpc: vpc-b
    StringNotEquals:
      OrgID: o-1
```

Fig. 1. An example AWS policy

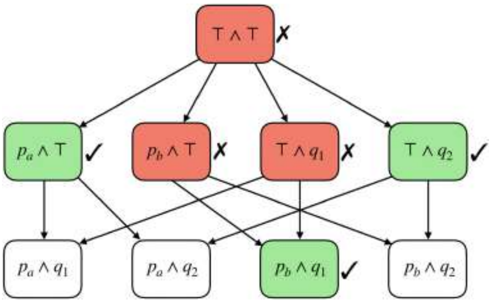


Fig. 2. Stratified abstraction search tree

In this paper, we present a new approach to help users understand whether their policy is correct, by *abstracting* the policy into a compact set of positive and declarative statements that precisely summarize *who* has access to a resource. Users can review the summary to decide whether the policy grants access according to their intentions. The key challenge to computing such summaries is the combinatorial blowup in the number of possible requests, which comprise the combination of user name and account, identifiers, hostnames, IP addresses and so on. Our key insight is that we can make summarization tractable via *stratified predicate abstraction*, which allows us to collapse many equivalent (concrete) requests into a single (abstract) *finding*. To this end, we introduce a new algorithm for computing stratified abstractions of policies, yielding a set of findings that are *sound*, *i.e.* which include all possible requests that can be granted access, and *precise*, *i.e.* where the findings are as specific as possible.

We have implemented stratified abstraction and deployed it as the engine powering AWS's recently launched IAM Access Analyzer service, which helps users reason about the semantics of their policy configurations. We present an empirical evaluation of our method over a large set of real-world IAM policies. We show that IAM Access Analyzer generates a sound, precise, and *compact* set of findings for complex policies, taking less than a second per finding. Thus, our results show how key ideas like SMT solving and predicate abstraction [1,5], can be used not just to *verify* computing systems, but to precisely *explain* their behavior to users.

2 Overview

AWS access control policies specify *who* has access to a given resource, via a set of `Allow` and `Deny` statements that grant and prohibit access, respectively. Figure 1 shows a simplified policy specifying access to a particular resource. This policy uses conditions based on which network (known as a VPC) the request originated from and which organizational Amazon customer (referred to by an Org ID) made the request. The first statement *allows* access to any request whose `SrcVpc` is either `vpc-a` *or* `vpc-b`. The second statement *allows* access to any request whose `OrgId` is `o-2`. However, the third statement *denies* access from `vpc-b` *unless* the `OrgId` is `o-1`.

Crucially, for each request, access is granted only if: (a) *some* `Allow` statement matches the request, and (b) *none* of the `Deny` statements match the request. Consequently, it can be quite tricky to determine what accesses are allowed by a given policy. First, individual statements can use regular expressions, negation, and conditionals. Second, to know the effect of an allow statement, one must consider all possible deny statements that can *overlap* with it, *i.e.* can refer to the same request as the allow. Thus, policy verification is not *compositional*, in that we cannot determine if a policy is "correct" simply by *locally* checking that each statement is "correct". Instead, we require a *global* verification mechanism, that simultaneously considers all the statements and their subtle interactions, to determine if a policy grants only the intended access.

As policies organically grow and become more complex and baroque, the ultimate question that users have is: "is my policy correct?" Of course, this *specification* problem has bedeviled formal methods from the day they were invented. In our context: how does the security analyst know whether the policy is, in fact not too strict or too permissive? Zelkova [2] is already used by users of Amazon's Simple Storage Service (S3) to determine whether any of their "data buckets" are *publicly* accessible. More generally, the AWS Config service provides templated Zelkova checks that can be filled in by users to validate their policies. Some advanced users even use the Zelkova service directly, asking their own questions about policies. While all of the above are useful, formal policies and formal analysis remains difficult to use, as the user must have sufficient technical sophistication to: (1) *intuit* the criteria important to them, (2) *formalize* the above in the query language of Zelkova, and (3) *interpret* the results returned by the tool. Ultimately, to answer "is this policy correct?", the tool must *help the user understand* what "correct" means in their particular context.

2.1 Approach

The core contribution of this work is to change the question from *"is this policy correct?"* to *"who has access?"*. The response to the former is a Boolean while the response to the latter is a set of *findings*. There are several key requirements that findings must meet to be useful in the context of analyzing security policies and answering the question *"who has access?"*.

Sound. Users need confidence that findings *summarize* a policy. In particular, we must ensure that *every* access allowed by the policy is represented by *some* finding. This over-approximation crucially enables compositional reasoning about the policy: if a user deems that *each* finding is safe, then she may rest assured that the *entire* policy is safe.

Precise. Users require that findings be *specific*. A finding of "everybody has access" is a sound and over-approximate summary of every policy, but is only useful if the policy allows everyone access. Instead, we want findings that adhere closely to the accesses allowed by the policy, and do not report false-alarms that say certain identities have access when that is not, in fact, the case.

Compact. Users require that the set of findings be *small*. For example, we could simply *enumerate* all the different kinds of requests that have access, but such a list would typically be far too large to manually inspect. Instead, we require that the findings be a compact representation of who has access, while still ensuring soundness and precision.

Example. For example, the policy in Fig. 1 can be summarized through a set of three findings, that say that access is granted to a request iff:

- Its `SrcVpc` is `vpc-a`, *or*,
- Its `OrgId` is `o-2`, or,
- Its `SrcVpc` is `vpc-b` *and* its `OrgId` is `o-1`.

The findings are sound as no other requests are granted access. The findings are precise as in each case, there are requests matching the conditions that are granted access.[1] Finally, the findings compactly summarize the policy in three positive statements declaring *who* has access.

2.2 Solution: Computing Findings via Stratified Abstraction

Next, we describe an informal overview of our algorithm for computing the findings, by building it up in three stages.

1: Concrete Enumeration. One approach to synthesize findings would be to (1) *enumerate* possible requests, (2) *query* Zelkova to filter out the requests that do not have access, and (3) *return* the remainder as findings. Such an approach is guaranteed to be both sound and precise. However, real-world policies comprise many fields, each of which have many possible values. For example, there are 10^{12} (currently) possible AWS account numbers and 2^{128} possible IPv6 addresses. Enumerating all possible requests is computationally *intractable*, and even if it were, the resulting set of findings is far too large and hence *useless*.

2: Predicate Abstraction. We tackle the problem of summarizing the super-astronomical request-space by using *predicate abstraction*. Specifically, we make a syntactic pass over the policy to extract the set of constants that are used to constrain access, and we use those constants to generate a family of predicates

[1] The finding "`OrgId` is `o-2`" also includes some requests that are not allowed, *e.g.* when `SrcVpc` is `vpc-b`.

whose conjunctions compactly describe partitions of the space of all requests. For example, from the policy in Fig. 1 we would extract the following predicates

$$\begin{array}{lll} p_a \doteq \mathsf{SrcVpc} = \texttt{vpc-a}, & p_b \doteq \mathsf{SrcVpc} = \texttt{vpc-b}, & p_\star \doteq \mathsf{SrcVpc} = \star, \\ q_1 \doteq \mathsf{OrgId} = \texttt{o-1}, & q_2 \doteq \mathsf{OrgId} = \texttt{o-2}, & q_\star \doteq \mathsf{OrgId} = \star. \end{array}$$

The first row has three predicates describing the possible value of the `SrcVpc` of the request: that it equals `vpc-a` or `vpc-b` or some value other than `vpc-a` and `vpc-b`. Similarly, the second row has three predicates describing the value of the `OrgId` of the request: that it equals `o-1` or `o-2` or some value other than `o-1` and `o-2`.

We can compute findings by enumerating all the *cubes* generated by the above predicates, and querying ZELKOVA to determine if the policy allows access to the requests described by the cube. For example, the above predicates would generate the cubes shown in Fig. 3. We omit trivially inconsistent cubes like $p_a \wedge p_b$ which correspond to the empty set of requests. Next to each cube, we show the result of querying ZELKOVA to determine whether the policy allows access to the requests described by the cube: ✓(resp. ✗) indicates requests are allowed (resp. denied).

$\boxed{p_a \wedge q_1}$	✓	$\boxed{p_a \wedge q_2}$	✓	$\boxed{p_a \wedge q_\star}$	✓
$\boxed{p_b \wedge q_1}$	✓	$\boxed{p_b \wedge q_2}$	✗	$\boxed{p_b \wedge q_\star}$	✗
$\boxed{p_\star \wedge q_1}$	✗	$\boxed{p_\star \wedge q_2}$	✓	$\boxed{p_\star \wedge q_\star}$	✗

Fig. 3. Cubes generated by the predicates $p_a, p_b, p_\star, q_1, q_2, q_\star$ generated from the policy in Fig. 1 and the result of querying ZELKOVA to check if the requests corresponding to each cube are granted access by the policy.

Finally, we can translate each *allowed* cube into a finding, yielding five findings. While this set of findings is sound and precise, it suffers in two ways. First, real-world policies have many different fields, and hence, enumerating-and-querying each cube can be quite slow. Second, the result is not compact. The same information is more succinctly captured by the set of three findings in Sect. 2.1 which, for example, collapses the three findings in the top row to a single finding, "`SrcVpc` is `vpc-a`."

3: Stratified Abstraction. The chief difficulty with enumerating all the cubes *greedily* is that we end up eagerly *splitting-cases* on the values of fields when that may not be required. For example, in Fig. 3, we split cases on the possible value of `OrgId` even though it is irrelevant when `SrcVpc` is `vpc-a`. This observation points the way to a new algorithm where we *lazily* generate the cubes as follows. Our algorithm maintains a *worklist* of minimally refined cubes. At each step, we (1) ask ZELKOVA if the cube allows an access that is not covered by any of its refinements, (2) if so, we add it to the set of findings; and (3) if not, we refine the cube "point-wise" along the values of each field individually and add the results to the worklist. The above process is illustrated in Fig. 2.

- ***Level 1.*** The worklist is initialized with $\top \wedge \top$ which represents the cube where we *don't care* about the value of either `SrcVpc` or `OrgId`, *i.e.* which represents *every* possible request. Zelkova determines that every access allowed by this cube and by the policy are covered by one of the refinements of this cube (the second level of the tree). Thus this $\top \wedge \top$ finding is not essential, and we can find more precise findings. We indicate this by the red shade and the ✗. Next, we *refine* the above cube point-wise, by considering the two sub-cubes $p_a \wedge \top$ and $p_b \wedge \top$ which respectively represent the requests where `SrcVpc` is either `vpc-a` or `vpc-b` (and `OrgId` could be any value), and, the two sub-cubes $\top \wedge q_1$ and $\top \wedge q_2$ which respectively represent the requests where `OrgId` is either `o-1` or `o-2` (and `SrcVpc` could be any value). These refined cubes are added to the worklist and considered in turn.
- ***Level 2.*** Zelkova determines that there are requests allowed by $p_a \wedge \top$ and $\top \wedge q_2$ which are not covered by any of their refinements, hence those are shaded green and have a ✓. However, Zelkova rejects $p_b \wedge \top$ and $\top \wedge q_1$ as anything allowed by them is allowed by one of their refinements. Now we further refine the rejected cubes, but can *omit* considering the cubes $p_a \wedge q_1$, $p_a \wedge q_2$ and $p_b \wedge q_2$ in the unshaded boxes, as each of those is covered or subsumed by one of the two accepted cubes.
- ***Level 3.*** Hence, we issue one last Zelkova query for $p_b \wedge q_1$ which indeed allows a request which is not covered by any of its refinements (as it has none). Finally, we gather the set of accepted cubes, *i.e.* those in the green shaded boxes, and translate those to the findings described in Sect. 2.1.

3 Algorithm

Next, we formalize our algorithm for computing policy summaries and show how it yields findings that are sound and precise. In Sect. 4 we demonstrate how our algorithm yields compact results for real-world policies..

3.1 Policies and Findings

Requests. Let $K = \{k_1, \ldots, k_n\}$ be a set of *keys*. Let $V_k = \{v_1, \ldots\}$ be a (possibly infinite) set of *values* for the key k. A *request* r a mapping from keys k to values in V_k. For example, the request r_1 maps the keys Principal, SrcIP, and OrgID as:

$$r_1 = \{\mathsf{Principal} \mapsto \mathtt{123:user/A},\ \mathsf{SrcIP} \mapsto \mathsf{192.0.2.3},\ \mathsf{OrgID} \mapsto \mathsf{o\text{-}1}\}$$

Policies. A *policy* is a predicate on requests $p : r \rightarrow Bool$. The *denotation* of a policy p is the set of requests it allows:

$$\gamma(p) \doteq \{r \mid p(r) = True\}$$

Predicates. A *predicate* is a map $\phi : V_k \rightarrow \mathit{Bool}$. The *denotation* of a predicate is the set of values that satisfy the predicate:

$$\gamma(\phi) \doteq \{v \mid \phi(v) = \mathit{True}\}$$

We define a partial order on predicates, $\phi_1 \preceq \phi_2$ iff $\gamma(\phi_1) \subseteq \gamma(\phi_2)$. For example:

$$\begin{array}{rcl} \phi_{123}(v) & \doteq & \text{“}v\text{ is a principal in account 123”} \\ \phi_{ua}(v) & \doteq & \text{“}v\text{ is } \texttt{user-a} \text{ in account 123”} \\ \phi_{ub}(v) & \doteq & \text{“}v\text{ is } \texttt{user-b} \text{ in account 123”} \end{array}$$

Here we have $\phi_{ua} \preceq \phi_{123}$ and $\phi_{ub} \preceq \phi_{123}$ because users are a type of principal. The set of predicates must always contain $\top$ and must have the following property: for all ϕ_1, ϕ_2 either $\phi_1 \preceq \phi_2$, $\phi_2 \preceq \phi_1$, or $\gamma(\phi_1) \cap \gamma(\phi_2) = \emptyset$. This ensures the set of predicates for a given key can be tree-ordered.

Findings. A *finding* σ is a map from keys K to predicates Φ. The *denotation* of a finding σ is the set of requests where each key k is mapped to a value v in the denotation of $\sigma(k)$:

$$\gamma(\sigma) \doteq \{r \mid \forall k.r(k) \in \gamma(\sigma(k))\}$$

We represent a finite set of findings as $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$. The *denotation* of a set of findings is the union of the denotations the findings:

$$\gamma(\{\sigma_1, \ldots, \sigma_n\}) \doteq \gamma(\sigma_1) \cup \cdots \cup \gamma(\sigma_n)$$

3.2 Properties

Next, we formalize the key desirable properties of findings, *i.e.* that they be sound, precise, and compact, as *coverage*, *irreducibility*, and *minimality* respectively.

Coverage. A set of findings Σ *covers* a policy p if $\gamma(p) \subseteq \gamma(\Sigma)$. For example, the set Σ_1 containing the two findings

$$\Sigma_1 \ \doteq \ \{[\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto \top], [\mathsf{SrcVpc} \mapsto \top, \mathsf{OrgID} \mapsto q_2]\}$$

corresponding to the green boxes on level 2 of Fig. 2, *does not* cover the policy from Fig. 1, as it excludes the request whose SrcVpc is `vpc-b` and OrgID is `o-1`. However, Σ_2 below *does* cover the policy as it includes all requests that are granted access.

$$\Sigma_2 \ \doteq \ \Sigma_1 \cup \{[\mathsf{SrcVpc} \mapsto p_b, \mathsf{OrgID} \mapsto q_1]\}$$

Reducibility. A finding σ refines another finding σ', written $\sigma \sqsubseteq \sigma'$ if *for each* key k we have $\sigma(k) \preceq \sigma'(k)$. A finding σ refines a set of findings Σ, written

$\sigma \sqsubseteq \Sigma$ if σ refines *some* $\sigma' \in \Sigma$. Note that $\sigma \sqsubseteq \sigma'$ implies $\gamma(\sigma) \subseteq \gamma(\sigma')$. We say that a finding σ is *irreducible* for a policy p if

$$\exists r \in \gamma(p) \cap \gamma(\sigma).\ \forall \sigma' \sqsubset \sigma.\ r \notin \gamma(\sigma').$$

That is, σ is irreducible if it contains some request that is excluded by all its proper refinements. For example, the finding $[\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto \top]$ is irreducible as it contains a request $[\mathsf{SrcVpc} \mapsto \mathsf{vpc\text{-}a}, \mathsf{OrgID} \mapsto \mathsf{o\text{-}3}]$ that is excluded by its refinements $[\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto q_1]$ and $[\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto q_2]$. Note that irreducibility is inherently tied to the available predicates, Φ.

Minimality. A set of findings Σ is *minimal* if the denotation of each $\Sigma' \subset \Sigma$ is strictly contained in the denotation of Σ. For example, the set

$$\{[\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto \top], [\mathsf{SrcVpc} \mapsto p_a, \mathsf{OrgID} \mapsto q_1]\}$$

is *not* minimal as the subset containing just the first finding denotes the same set of requests, but, the set containing either finding individually *is* minimal.

3.3 Algorithm

Given a policy p and a finite set of partially ordered predicates Φ, our goal is to produce a minimal covering of p comprising only irreducible findings.

Access Oracle. Our algorithm is built using an *access oracle* that takes as input a policy p and a finding σ and returns Some iff some request described by σ is allowed by p, and None otherwise.

$$\mathsf{CanAccess}(p, \sigma) = \begin{cases} \mathsf{Some} & \text{if } \gamma(\sigma) \cap \gamma(p) \neq \emptyset \\ \mathsf{None} & \text{if } \gamma(\sigma) \cap \gamma(p) = \emptyset \end{cases}$$

```
def AccessSummary($p$:$P$) -> [$\Sigma$]:
  $\sigma_\top$ = $\lambda k \to \top$
  $wkl$ = queue([$\sigma_\top$])
  $res$ = []
  while $wkl \neq \emptyset$:
    $\sigma$ = $wkl$.deque()
    if CanAccess($p$,Reduce($\sigma$)) == Some:
      $res$ += [$\sigma$]
    else:
      $wkl$ += [$\sigma'$ | $\sigma' \in$ Refine($\sigma$), $\sigma' \not\sqsubseteq res$]
  return $res$
```

Fig. 4. Algorithm to compute a minimal set of irreducible findings that cover policy p.

Dominators. We define the *immediately dominates* set of $\phi \in \Phi$ as the set of elements strictly smaller than ϕ but unrelated to each other:

$$\mathsf{idom}(\phi) \doteq \{\phi' \mid \phi' \prec \phi \text{ and } \forall \phi''.\neg(\phi' \prec \phi'' \prec \phi)\}$$

Reducing a Finding. The procedure ReducePred (resp. Reduce) takes as input a predicate ϕ (resp. finding σ) and strengthens it to *exclude* all the requests that are covered by the refinements of ϕ (resp. σ):

```
def ReducePred($\phi$:$\Phi$) -> $\Phi$:
  $\phi_1, \ldots, \phi_n$ = idom($\phi$)
  return $\phi \wedge \neg\phi_1 \wedge \cdots \wedge \neg\phi_n$
```

```
def Reduce($\sigma$:$\Sigma$) -> $\Sigma$:
  $\sigma'$ = $\lambda k \rightarrow$ ReducePred($\sigma(k)$)
  return $\sigma'$
```

Intuitively, Reduce allows us to determine if a finding is irreducible.

Lemma 1. *σ is irreducible iff* $\gamma(\mathsf{Reduce}(\sigma)) \cap \gamma(p) \neq \emptyset$.

Refining a Finding. The procedure Refine takes as input a finding σ and returns the set of findings obtainable by *individually* refining one value of σ.

```
def Refine($\sigma$:$\Sigma$) -> [$\Sigma$]:
  return [$\sigma[k \mapsto \phi']$ | $k \in K$, $\phi' \in$ idom($\sigma(k)$)]
```

If a finding σ is reducible, we will use Refine to *split* it into more precise findings.

Lemma 2. *Let σ be reducible for p. Then* $\gamma(\sigma) \cap \gamma(p) = \gamma(\mathsf{Refine}(\sigma)) \cap \gamma(p)$.

Summarizing Access. The procedure AccessSummary (Fig. 4) takes as input a policy p and returns a minimal set of irreducible findings *res* that covers p. The procedure maintains a queue *wkl* comprising a *frontier* of findings that are to be explored. The queue is initialized with the trivial finding $\sigma_\top$ that maps each key to $\top$. It then iteratively picks an element from the queue, checks if it is an irreducible finding, and if so, adds it to the result set *res*. If not, it computes the finding's refinements and adds those to *wkl*. The process repeats till the queue is empty. The algorithm maintains three loop invariants: (1) $wkl \cup res$ covers p; (2) Each finding in *res* is irreducible; (3) *res* is minimal. Consequently, the algorithm terminates with a minimal set of irreducible findings that covers p. Note, the worklist is a queue so that if $\sigma_1 \sqsubset \sigma_2$ the algorithm will consider σ_2 before σ_1.

Theorem 1. *Let $\Sigma = \mathsf{AccessSummary}(p)$. Then (1) Σ covers p, (2) each $\sigma \in \Sigma$ is irreducible, and (3) Σ is minimal.*

4 Implementation and Evaluation

The algorithm AccessSummary is implemented in the IAM Access Analyzer feature launched on Dec 2, 2019 [10]. The ZELKOVA tool [2] is used as the access oracle for the algorithm. Access Analyzer monitors the relevant resource policies in an account and re-runs the algorithm on any changes. Findings are presented to the user through a web console and through APIs. Users can *archive* findings that represent intended access to the resource. For unintended findings, Access Analyzer links to the relevant policy that users can edit to remove that access. Access Analyzer will automatically run on the changed policy and any findings that are no longer relevant will be set to a *resolved* state. By monitoring any existing or new *active* findings, users can ensure their polices grant only the intended access.

Evaluation Metrics. We evaluate our algorithm along two dimensions: (1) "how efficient is the algorithm at generating findings?" and (2) "how effective are the generated findings at simplifying the complexity of a policy?". As our algorithm solves a new problem, we do not have an external basis for comparison. Instead, we compare the algorithm against the state space it operates over. To this end, for each policy, we define the following measures:

– **size** is the size of the set of all possible findings for the policy.
– **findings** is the number of findings produced by the algorithm.
– **queries** is the number of SMT queries made by the algorithm.
– **runtime** is the total runtime of the algorithm.

Note that **findings** $\leq$ **queries** $\leq$ **size**, as each query generates at most one finding and we query each possible finding at most once.

Benchmarks. We randomly selected 1,387 policies from a corpus of in-use policies. As we are interested primarily in difficult policies, we filtered out all policies that had **size** less than 10. That left 165 policies. Each policy was evaluated on a 2.5 GHz Intel Core i7 with 16 GB of RAM. The runtime per finding (**runtime/findings**) was less than 430ms for all policies except one outlier at 2,267 ms. The 165 policies ranged in size from 56 to 810 lines of pretty-printed JSON with a median size of 91 lines.

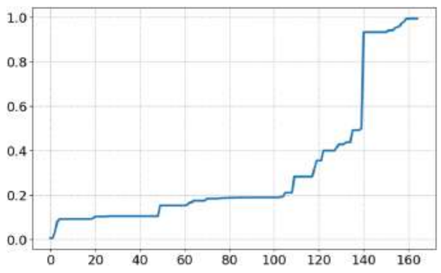


Fig. 5. Actual findings vs. search space

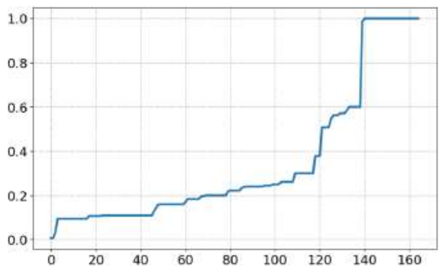


Fig. 6. Actual queries vs. search space

Results. Figures 5 and 6 show the number of findings and queries, respectively, compared to the overall search space. Both graphs are sorted to be monotonic, *i.e.* the x-axes are different. Figure 5 shows to what degree the findings simplify the policy, with smaller numbers being better. This measure will always be between 0 and 1 since 0 $\leq$ **findings** $\leq$ **size**. We see that 85% of policies achieve a ratio of 0.5 or better, and 64% achieve a ratio of 0.2 or better. Figure 6 shows how efficient the algorithm is in exploring its state space, with smaller numbers being better. This measure is between 0 and 1 as 0 $\leq$ **queries** $\leq$ **size**. The algorithm explores the entire search space for only 15% of the policies, with a median ratio of 0.22.

5 Related Work

The majority of tools available for access policy analysis are based on log analysis or syntactic pattern matching, which are both imprecise (*i.e.* fail to account for the complex logic in AWS policies) and unsound (*i.e.* fail to check for all requests) and hence, can take months to discover that resources are susceptible to potentially unintended access. Most formal methods based work has focused on securing individual pieces of cloud infrastructure via low-level proofs of software correctness *e.g.* Ironclad [6]. Cloud Contracts [4] are requirements over network access control lists and routing tables. Cloud Contracts are verified using the SecGuru tool [7] that compares network connectivity policies using the SMT theory of bit vectors. In contrast, our work answers a larger question about the entire enterprise-level security posture using a series of Zelkova queries [2]. The Fireman system [11] shows how to use Binary Decision Diagrams to analyze access control lists (ACL) in firewall configurations. The ACL configuration language is more restricted than IAM's and the tool is limited to a fixed set of queries about which accesses (packets) are allowed. Most closely related to our work is the Margrave system [9] which encodes firewall policies as propositional logic formulas, and then use SAT solvers to answer queries about the policies. Margrave introduces the notion of *scenario finding*, and shows how to produce an exhaustive set of scenarios that *witness* the queried behavior. The IAM policy language is significantly richer, and hence, enumerating scenarios is computationally intractable, which led us to the develop stratified abstraction as a means of summarizing policy semantics, thereby providing analysts comprehensive visibility into the accessibility of resources, helping detect misconfigurations, and ensuring that updates indeed fix the potential for unintended accesses.

References

1. Agerwala, T., Misra, J.: Assertion graphs for verifying and synthesizing programs. Technical report, University of Texas at Austin, USA (1978)
2. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9. IEEE (2018)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bjørner, N., Jayaraman, K.: Checking cloud contracts in microsoft azure. In: Natarajan, R., Barua, G., Patra, M.R. (eds.) ICDCIT 2015. LNCS, vol. 8956, pp. 21–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14977-6_2
5. Grumberg, O. (ed.): CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6
6. Hawblitzel, C., et al.: Ironclad apps: end-to-end security via automated full-system verification. OSDI **14**, 165–181 (2014)
7. Jayaraman, K., Bjorner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Technical report MSR-TR-2014-102, Microsoft Research (2014)

8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The margrave tool for firewall analysis. In: Proceedings of the 24th International Conference on Large Installation System Administration, LISA 2010, pp. 1–8. USENIX Association, USA (2010)
10. West, B.: AWS news blog, December 2019. https://aws.amazon.com/blogs/aws/identify-unintended-resource-access-with-aws-identity-and-access-management-iam-access-analyzer/
11. Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., Mohapatra, P.: FIREMAN: a toolkit for firewall modeling and analysis. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, California, USA, May 21–24, pp. 199–213 (2006). https://doi.org/10.1109/SP.2006.16

Synthesis of Super-Optimized Smart Contracts Using Max-SMT

Elvira Albert[1,2], Pablo Gordillo[2](✉), Albert Rubio[1,2], and Maria A. Schett[3]

[1] Instituto de Tecnología del Conocimiento, Madrid, Spain
[2] Complutense University of Madrid, Madrid, Spain
pabgordi@ucm.es
[3] University College London, London, U.K.

Abstract. With the advent of smart contracts that execute on the blockchain ecosystem, a new mode of reasoning is required for developers that must pay meticulous attention to the *gas* spent by their smart contracts, as well as for optimization tools that must be capable of effectively reducing the gas required by the smart contracts. Super-optimization is a technique which attempts to find the best translation of a block of code by trying all possible sequences of instructions that produce the same result. This paper presents a novel approach for super-optimization of smart contracts based on Max-SMT which is split into two main phases: (i) the extraction of a *stack functional specification* from the basic blocks of the smart contract, which is simplified using rules that capture the semantics of the arithmetic, bit-wise, relational operations, etc. (ii) the *synthesis of optimized blocks* which, by means of an efficient Max-SMT encoding, finds the bytecode blocks with minimal gas cost whose stack functional specification is equal (modulo commutativity) to the extracted one. Our experimental results are very promising: we are able to optimize 55.41 % of the blocks, and prove that 34.28 % were already optimal, for more than 61 000 blocks from the most called 2500 Ethereum contracts.

1 Introduction

Open-source software that leverages on the blockchain ecosystem is known as *smart contract*. Smart contracts are not necessarily restricted to the classical concept of contracts, but can be any kind of program that executes on a blockchain or distributed ledger. A smart contract can be regarded as a collection of secured stored functions whose execution and effects (e.g., the transfer of some value between parties) cannot be manipulated. This is because all records of the transactions must be stored on a public and decentralized blockchain that avoids the

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM project P2018/TCS-4314 and by the UCM CT27/16-CT28/16 grant.

pitfalls of centralization. While Bitcoin [21] paved the way for cryptocurrencies and for the popularity of the blockchain technology, Ethereum [25] showed the full potential of blockchains by allowing developers to run their decentralized applications on top of their platform. The Ethereum Virtual Machine (EVM) is capable of running smart contracts coded by Ethereum developers that have the potential of replacing all sorts of legal, financial and social agreements, e.g., can be used to fulfill employment contracts, execute bets and wagers, etc.

On the Ethereum blockchain platform, as well as in other emerging blockchains equipped with a smart contract programming language (e.g., Tezos [1], Zilliqa [24], Facebook's Libra [23]), *gas* refers to the fee, or pricing value, required to successfully conduct a transaction or to execute a smart contract. Gas is priced in a sub-unit of the cryptocurrency—in Ethereum in *gwei*, a sub-unit of its *Ether* cryptocurrency. The EVM specification [25] provides the *gas model*, i.e., a precise definition of the gas consumption for each EVM bytecode instruction. The EVM is a simple stack-based architecture: computation on the EVM is done using a stack-based bytecode language; the word size of the machine is 256-bits (32-bytes), and this is also the size of a stack item. The proposer of a transaction allots an amount of gas (known as gas limit) to carry out the execution. If the transaction exceeds the allotted gas limit, an *out-of-gas* exception is raised, interrupting the current execution. The rationale of gas metering is three-fold: first, a gas-metered execution puts a cap on the number of operations that a transaction can execute and prevents attacks based on non-terminating executions; second, paying for gas at the moment of creating the transaction does not allow the proposer to waste other parties' (aka *miners*) computational resources; third, gas fees discourage users to overuse replicated *storage*, which is an expensive and valuable resource in a blockchain-based consensus system.

Optimization of smart contracts has thus a clear optimization target: gas usage, as both computational and storage costs are accounted within the gas cost of each of the EVM instructions. Indeed, reducing gas costs of smart contracts is a problem of utmost relevance in the blockchain ecosystem, as there are normally between half a million and a million transactions a day. The cost of a transaction in Ethereum ranges from cents to few dollars, except in certain peak periods that has been ten or a hundred times more. In order to provide an idea of the impact of gas saving techniques, we have estimated that the money spent in transactions (excluding the intrinsic gas cost) from 2017 to 2019 is around 157 Million dollars[1]. Thus, optimizing programs in an energy-saving way is essential in general, but it is even more so in the blockchain ecosystem. The Solidity[2] documentation [13], and posterior documents (e.g., [9,19]), identify gas-costly patterns and propose replacements with gas-efficient ones. Adopting these guidelines requires a deep understanding of EVM instructions and the gas consumption for the different operations. Compilers for Solidity also try to optimize the bytecode for minimizing its gas consumption (e.g., the flag optimize of the

[1] The data is taken from [3] using the gas spent by transactions and the average *gwei* and Ether exchange rate per day.

[2] It is the most popular programming language for writing Ethereum smart contracts.

`solc` compiler optimizes storage of large constants and the `dispatch` routine, with the goal of saving gas).

Even when the guidelines are followed and the `optimize` flag is used, the compiled EVM code is not always as efficient as desired. Super-optimization [17] is a technique proposed over 30 years ago which attempts to find the best translation of a block of code using exhaustive search to try all possible sequences of instructions that produce the same result. As an exhaustive search problem, it is computationally extremely demanding. The work in [15] proposed the idea of "unbounded" super-optimization that consists in shifting the *search* for the target program into the solver. Recently, unbounded super-optimization has been applied to Ethereum bytecode [20] for *basic block* optimization (i.e., optimizations are made inside a basic block formed by a sequence of instructions without any `JUMP` operation in the middle). The experimental results in [20] confirm the extreme computational demands of the technique (e.g., the tool times out in 92% of the blocks used in their evaluation). This is a severe limitation for the use of the technique, and the problem of finding the optimal code for an EVM block still remains very challenging. The complexity stems mainly from three sources: First, the problem is expressed in the theory of bit-vector arithmetic with bit-width size of 256, which is a challenging width size for most SMT solvers. Second, expressing the problem involves an exists-forall quantification, since we want to find an assignment of instructions that works for all values in the initial stack. Third, since we look for the gas-optimal code, the problem is not a satisfaction problem but rather an optimization problem.

Contributions. This paper proposes a novel method for gas optimization of smart contracts which is based on synthesizing optimized EVM blocks using Max-SMT. The main novel features that distinguish our work from previous approaches, that attack the same or a similar problem [15,20], are:

1. *Stack functional specification.* Our method takes as input an EVM bytecode and first obtains from it a stack functional specification (SFS) of the input and output operational stacks for each of the blocks of the control-flow graph (CFG) for the bytecode by using symbolic execution. The SFS determines thus the target stack that the block has to compute and is simplified using a set of rules that capture a great part of the semantics of the arithmetic, bit-wise, relational, etc., EVM operations which are relevant for gas optimization.
2. *Synthesis problem using SMT.* We approach optimization as a synthesis problem in which an SMT solver is used to synthesize optimal EVM bytecode which, for the input stack given in the functional specification, produces the target stack determined by the specification. We present a very efficient encoding that, in contrast to the previous attempts, uses only existential quantification in a very simple fragment of integer arithmetic. According to our evaluation, its simplicity greatly improves the performance of the SMT solvers while accuracy is kept as we cover the main possible optimizations. Importantly, only the semantics of the stack operations (`PUSH`, `DUP`, `SWAP`, etc.) is encoded, while all other operations are treated as uninterpreted functions.

3. *Use of Max-SMT.* We encode the optimization problem using Max-SMT, by adding soft constraints that encode the gas cost of the selected instructions, by adding the needed weights. This allows us to take advantage of the features given by recent Max-SMT optimizers that can improve the search.
4. *Experiments.* We report on syrup, an implementation of our approach, and evaluate it on (i) the same data set used for evaluating the tool ebso from [20] and, (ii) on 128 of the most called contracts on the Ethereum blockchain. Our results are very promising: while ebso timed out in 92.12 % of the blocks in (i), we only time out in 8.64 % and obtain gains that are two orders of magnitude larger than ebso. These results show that we have found the right balance between what is optimized by means of symbolic execution and symbolic simplification using rules and what is encoded as a Max-SMT problem. Moreover, for set (ii), we obtain gas savings of 0.59% of the total gas. Assuming that these savings are uniformly distributed, it would amount nearly to 1 Million dollars from 2017 to 2019.

While the purpose of superoptimization is to optimize at the level of basic blocks (intra-block), our approach to synthesize EVM code from a given SFS can be applied also in a richer optimization framework that enables the optimization of multiple basic blocks (inter-block). For this purpose, the framework should be extended to include branching instructions (which in the SMT encoding can be handled with uninterpreted functions as well) and, besides, additional components would be required, e.g., in the context of EVM we would need to resolve the jumping addresses, and to ensure that there are no additional incoming jumps to intermediate blocks that are being merged by the optimizer. Inter-block optimization is especially interesting in the context of smart contracts to gain storage-related gas, since the optimizations that can be achieved locally for the storage are quite limited as explained in Sect. 6.

```
pragma solidity ^0.4.25;
contract addExp{
  function ae(uint x3, uint x2, uint x1,
    uint x0) returns (uint){
      uint x = x3+x2;
      uint y = x1+x0;
      return x**y;    //EXP operation
  }
}
```

```
JUMPDEST
PUSH1 0x00
DUP1
PUSH1 0x00
DUP6
DUP8
ADD
SWAP2
POP
DUP4
DUP6
ADD
SWAP1
POP
DUP1
DUP3
EXP
SWAP3
POP
POP
POP
SWAP5
SWAP4
POP
POP
POP
POP
JUMP
```

Fig. 1. Solidity code (left). Under-optimized EVM bytecode using solc (right).

2 Overview: Optimal Bytecode as a Synthesis Problem

This section provides a general overview of our method for synthesizing super-optimized smart contracts from given EVM bytecode. We use the motivating example in Fig. 1 whose Solidity source code contract appears to the left and the EVM bytecode generated by the `solc` compiler appears to the right. Solidity is an object-oriented, high-level language that is statically typed, supports inheritance, libraries and user-defined types, among other features. It is designed to target the EVM. As it can be observed in the example the EVM bytecodes that operate on the stack (i.e., `DUP`, `SWAP`, `ADD`, `AND`, etc.) are standard operators. In the following, we refer as *stack operations* only to `DUP`, `PUSH`, `SWAP` and `POP`, which modify the stack without performing computations. The EVM has also bytecodes to access persistent data stored in the contract's storage (`SLOAD` and `SSTORE`), to access data stored in the local memory (`MLOAD` and `MSTORE`), bytecodes that jump to a different code address location (`JUMP`, `JUMPI`), bytecodes for calling a function on a different contract (`CALL`, `DELEGATECALL`, `CALLCODE` and `CALLSTATIC`), to write a log (`LOG`), to access information about the blockchain and transaction (`GAS`, `CALLER`, `BLOCKHASH`, etc.) and copy information related to an external call (`CODECOPY`, `RETURNDATACOPY`, etc.). However, as we explain in the coming sections, our approach is based on optimizing the operations that modify the stack as we have a great coverage of all potential bytecode optimizations while we still remain scalable, i.e., we do not optimize those bytecodes whose effects are not reflected in the stack, e.g., `MSTORE`, `SSTORE`, `LOG1` or `EXTCODECOPY`. The gas consumed by this bytecode (excluding the `JUMPDEST` and `JUMP` opcodes that cannot be optimized and are thus not accounted in the examples) is 76. As specified in [25], the operations from the so-called *base* family (like `POP`) have cost 2, the operators from the *verylow* family (like `PUSH`, `SWAP`, `ADD`) cost 3, operators from the *low* family (like `MUL`, `DIV`) cost 5, and so on.

2.1 Extracting Stack Functional Specifications from EVM Bytecode

Our method takes as input the set of blocks that make up the control flow graph (CFG) of the bytecode. The first step is, for each of the blocks, to extract from it a *stack functional specification* (SFS) from which the super-optimized bytecode will be synthesized. The SFS is a functional description of the initial stack when entering the block and the final stack after executing the block, which instead of using bytecode instructions to determine how the final stack is computed, is defined by means of *symbolic first-order terms* over the initial stack elements. The SFS for our running example is shown in Fig. 2.

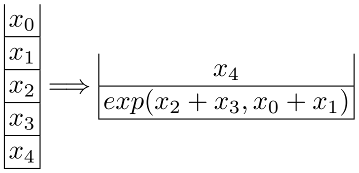


Fig. 2. Initial and final stack

As can be observed, it consists of an initial stack shown at the left which simply determines what the size of the input stack to the block is and assigns a symbolic variable as identifier to each stack position (e.g., the initial stack contains five elements named $x_0, \dots, x_4$); while the output stack contains two elements: x_4 at the top, and the symbolic term $exp(x_2+x_3, x_0+x_1)$ at the bottom. The output stack is obtained by symbolic execution of the bytecodes that operate on the stack, as it will be formalized in Sect. 3. The resulting expressions are then optimized by means of simplification rules based on the semantics of the non-stack operations (e.g., the neutral elements, double negations or idempotent operations are removed, operations on constants performed). This captures a relevant part of the semantics of the non-stack operators.

2.2 The Synthesis Problem

This section hints on how the generated bytecode will be, and on that the synthesis of optimal bytecode from the specification is challenging.

Example 1. From the SFS in Fig. 2, we know that we have to compute $x_0 + x_1$ and $x_2 + x_3$, but we have to decide which summation we compute first. On the left, we have the best bytecode (together with the stack evolution) when we first compute $x_2 + x_3$ and on the right when we first compute $x_0 + x_1$. Computing first one subexpression or the other has an impact on the consumed gas, since the bytecode on the left has a gas cost of 31 and the bytecode on the right has a gas cost of 25, which is indeed the optimum.

`SWAP3` $[x_3, x_1, x_2, x_0, x_4]$
`SWAP1` $[x_1, x_3, x_2, x_0, x_4]$
`SWAP2` $[x_2, x_3, x_1, x_0, x_4]$
`ADD` $[x_2 + x_3, x_1, x_0, x_4]$
`SWAP2` $[x_0, x_1, x_2 + x_3, x_4]$
`ADD` $[x_0 + x_1, x_2 + x_3, x_4]$
`EXP` $[(x_0 + x_1) ** (x_2 + x_3), x_4]$
`SWAP1` $[x_4, (x_0 + x_1) ** (x_2 + x_3)]$

`ADD` $[x_0 + x_1, x_2, x_3, x_4]$
`SWAP2` $[x_3, x_2, x_0 + x_1, x_4]$
`ADD` $[x_3 + x_2, x_0 + x_1, x_4]$
`SWAP1` $[x_0 + x_1, x_3 + x_2, x_4]$
`EXP` $[(x_0 + x_1) ** (x_2 + x_3), x_4]$
`SWAP1` $[x_4, (x_0 + x_1) ** (x_2 + x_3)]$

Both codes are far better than the original generated bytecode whose gas cost was 76. Besides, note that the cost of the two additions and the exponentiation is in total 16 (that necessarily has to remain), which means that the optimal code has used only 9 units of gas for the rest while the original code needed 60 units.

The next example shows that the optimal code is obtained when the subterms of the exponential are computed in the other order (compared to the previous example). Hence, an exhaustive search of all possibilities (with its associated computational demands) must be carried out to find the optimum.

Example 2. Let us now consider a slight variation of the previous example in which the functional specification is $[x_0, x_1, x_2, x_3] \Longrightarrow [x_3, (x_0{+}x_1) ** (x_0{+}x_2)]$. Now, on the left-hand side we have the best bytecode (together with the stack evolution) when we compute first $x_0 + x_2$ and on the right-hand side we have the best bytecode when we compute first $x_0 + x_1$.

`DUP1`	$[x_0, x_0, x_1, x_2, x_3]$	`DUP1`	$[x_0, x_0, x_1, x_2, x_3]$
`SWAP3`	$[x_2, x_0, x_1, x_0, x_3]$	`SWAP2`	$[x_1, x_0, x_0, x_2, x_3]$
`ADD`	$[x_2 + x_0, x_1, x_0, x_3]$	`ADD`	$[x_1 + x_0, x_0, x_2, x_3]$
`SWAP2`	$[x_0, x_1, x_2 + x_0, x_3]$	`SWAP2`	$[x_2, x_0, x_1 + x_0, x_3]$
`ADD`	$[x_0 + x_1, x_2 + x_0, x_3]$	`ADD`	$[x_2 + x_0, x_1 + x_0, x_3]$
`EXP`	$[(x_0 + x_1) ** (x_2 + x_0), x_3]$	`SWAP1`	$[x_1 + x_0, x_2 + x_0, x_3]$
`SWAP1`	$[x_3, (x_0 + x_1) ** (x_2 + x_0)]$	`EXP`	$[(x_1 + x_0) ** (x_2 + x_0), x_3]$
		`SWAP1`	$[x_3, (x_1 + x_0) ** (x_2 + x_0)]$

In this case the bytecode on the left has a gas cost of 28, which is indeed the optimum, and the bytecode on the right has a gas cost of 31. The original bytecode generated by `solc` has gas cost 74, so again the improvement is huge.

Both examples show that, in principle, even if we have the functional specification that guides the search, we have to exhaustively try all possible ways to obtain it, if we want to ensure that we have found the optimal bytecode.

2.3 Characteristics of Our SMT Encoding of the Synthesis Problem

Our approach to super-optimize blocks is based on restricting the problem in such a way that we have both a great coverage of most EVM code optimizations and we can propose an encoding in a simple theory where an SMT solver can perform efficiently. To this end, the key point is to handle all non-stack operations, like `ADD`, `SUB`, `AND`, `OR`, `LT`, as *uninterpreted bytecodes*. This allows us to simplify the encoding in two directions. First, by considering them as uninterpreted bytecodes we can avoid reasoning on the theory of bit-vectors with width 256. Second, and even more important, this allows us to express the problem in the existentially quantified fragment, avoiding the exists/forall alternation:

1. We start from the SFS by introducing fresh variables abstracting out all terms built with uninterpreted functions, in such a way that every fresh variable represents a term $f(a_1, \ldots, a_n)$, where every a_i is either a (256 bit) numeric value, a fresh variable, or an initial stack variable. We also have sharing by having a single variable for every term, e.g., $(x_0+1) ** (x_0+1)$, where x_0 is the top of the initial stack, is abstracted into $y_0 = \mathtt{EXP}_\mathtt{U}(y_1, y_1)$ and $y_1 = \mathtt{ADD}_\mathtt{U}(x_0, 1)$, where y_0 and y_1 are fresh variables and $\mathtt{EXP}_\mathtt{U}$ and $\mathtt{ADD}_\mathtt{U}$ are the uninterpreted bytecodes for exponentiation and addition, respectively.
2. Now, in order to avoid universal quantification, we take advantage of the fact that only values from 0 to $2^{256} - 1$ can be introduced in the stack by a `PUSH` opcode and hence only this range can appear in the SFS. Therefore, if we assign values from 2^{256} on to fresh variables and initial stack variables we avoid the confusion between themselves and all other values in the problem.

After these two key observations have been made, we fix the maximal number n of opcodes and highest size h of the stack that is allowed in a solution. This can be bound by analyzing the original code generated by the compiler. From this, we roughly encode the problem using variables $o_0, \ldots, o_{n-1}$ to express the operations of our code (together with variables $p_0, \ldots, p_{n-1}$ that encode the value

$0 \leq p_i \leq 2^{256}-1$ added to the stack when o_i is a `PUSH`), variables $s_0^i, \ldots, s_{h-1}^i$ to encode the contents of the stack before executing the operation o_i, where s_0^i is the top of the stack (we also use some Boolean variables to express the active part of the stack). Using this, we can encode the behavior of all stack operations: `POP`, `PUSH`, `DUP`, `SWAP` for all its versions (like `DUP1`, `DUP2`, ...). For the uninterpreted bytecodes f_u, we basically add for every abstraction $y = f_u(a_1, \ldots, a_m)$ assertions stating that if we have $a_1, \ldots, a_m$ at the top of the stack at step i (i.e., $s_0^i, \ldots, s_{m-1}^i$) and we take the operation f in o_i then in step $i+1$ we have $y, s_m^i, \ldots$ on the top of the stack. Again, as all fresh variables and initial stack variables have been replaced by values form 2^{256} on, there is no confusion with all other values.

As a final remark, we have also encoded the commutativity property of uninterpreted bytecodes representing the `ADD`, `MUL`, `AND`, `OR`, etc. This can be easily made by considering that the arguments can occur at the top of the stack in the two possible orders. Other properties like associativity are more difficult to encode and are left for future developments.

2.4 Optimal Synthesis Using Max-SMT

The last key element is how we encode the optimization problem of finding the bytecode with minimal gas cost. First, let us describe which notion of optimality we are considering. Our problem is defined as, given an SFS in which all occurring bytecodes there are considered uninterpreted and maybe commutative, we have to provide the bytecode with minimal gas cost whose SFS is equal modulo commutativity to the given one. From the encoding we have described in the previous section, we know that every solution to the SMT problem will have the same SFS as the given one. Hence, we only need to find the solution with minimal gas cost. In [20], this was made by implementing a loop on top of the SMT solving process which was calling the solver asking every time for a better solution in terms of gas, which was also encoded in the SMT problem. Such approach cannot be easily implemented in an incremental way using the SMT solver as a black box without the corresponding performance penalty.

Alternatively, we propose to encode the problem as a Max-SMT problem and hence, we can easily use any Max-SMT optimizer, like Z3 [12], Barcelogic [7] or (Opti)MathSAT [11], as a black box with an important gain in efficiency. The Max-SMT encoding adds to the previously defined SMT encoding some soft constraints, indicating which is the cost associated to choosing every family of operators. As mentioned, choosing an operator from the *base* family has cost 2, from the *verylow* 3, and so on. Then, the optimal solution is the solution that minimizes this cost, which can be obtained with a Max-SMT optimizer.

29 SSTORE	35 DUP2	41 DUP1	47 PUSH1 0x01
30 SWAP1	36 MSTORE	42 SWAP2	48 SWAP2
31 DUP5	37 PUSH1 0x20	43 SUB	49 SWAP1
32 SWAP1	38 ADD	44 SWAP1	50 POP
33 MLOAD	39 PUSH1 0x40	45 LOG2	51 JUMP
34 SWAP1	40 MLOAD	46 POP	

Block 1	Block 2		Block 3
30 SWAP1 31 DUP5 32 SWAP1 33 MLOAD 34 SWAP1 35 DUP2	37 PUSH1 0x20 38 ADD 39 PUSH1 0x40 40 MLOAD	41 DUP1 42 SWAP2 43 SUB 44 SWAP1	46 POP 47 PUSH1 0x01 48 SWAP2 49 SWAP1 50 POP

Fig. 3. CFG block of a real smart contract (top), and blocks generated to build the functional description of the EVM bytecode (bottom)

3 Stack Functional Specification from EVM Bytecode

The starting point of our work is the CFG of the EVM bytecode to be optimized. There are already a number of tools (e.g., EthIR [6], Madmax [14], Mythril [18] or Rattle [4]) that are able to compute the CFG from the bytecode of a given smart contract. Therefore, we do not need to formalize, neither to implement, this initial CFG generation step. Since there are bytecode instructions that we do not optimize, for each of the blocks of the provided CFG, we first perform a further block-partitioning that splits a basic block into the sub-blocks that will be optimized by our method as defined below. A basic block is defined as a sequence of EVM instructions without any `JUMP` bytecode.

Definition 1 (block-partitioning). *Given a basic block $B = [b_0, b_1, ..., b_n]$, we define its block-partitioning as follows:*

$$blocks(B) = \left\{ B_i \equiv b_i, \ldots, b_j \;\middle|\; \begin{array}{l} (\forall k. i < k < j, b_k \notin \mathit{Jump} \cup \mathit{Terminal} \cup \mathit{Split} \cup \\ \{\mathit{JUMPDEST}\}) \wedge (\ i{=}0 \vee b_{i-1} \in \mathit{Split} \cup \{\mathit{JUMPDEST}\}\ ) \wedge \\ (\ j{=}n \vee b_{j+1} \in \mathit{Jump} \cup \mathit{Split} \cup \mathit{Terminal}\ ) \end{array} \right\}$$

where

$$\begin{aligned} \mathit{Jump} &= \{\mathit{JUMP},\ \mathit{JUMPI}\} \\ \mathit{Terminal} &= \{\mathit{RETURN}, \mathit{REVERT}, \mathit{STOP}, \mathit{INVALID}\} \\ \mathit{Split} &= \{\mathit{SSTORE},\ \mathit{MSTORE},\ \mathit{LOGX},\ \mathit{CALLDATACOPY},\ \mathit{CODECOPY},\ \mathit{EXTCODECOPY}, \\ &\quad\ \mathit{RETURNDATACOPY}\} \end{aligned}$$

As it can be observed, the bytecodes whose effects are not reflected on the stack induce the partitioning and are omitted in the fragmented sub-blocks. These include the bytecodes that modify the memory, the storage or record a log, that belong to the *Split* set. Figure 3 shows a CFG block at the top and the blocks generated to build the functional description at the bottom. The original CFG block contains the bytecodes `SSTORE`, `MSTORE` and `LOG2`. Thus, it is split into three different blocks that do not contain these bytecodes.

$$
\begin{array}{lrl}
(1) & \tau(\mathcal{S}, \texttt{PUSHX}\ A) & = [A \mid \mathcal{S}] \\
(2) & \tau(\mathcal{S}, \texttt{DUPX}) & = [\mathcal{S}[0] \mid \mathcal{S}] \\
(3) & \tau(\mathcal{S}, \texttt{SWAPX}) & = \mathit{temp} = \mathcal{S}[0], \mathcal{S}[0] = \mathcal{S}[X], \mathcal{S}[X] = \mathit{temp} \\
(4) & \tau(\mathcal{S}, \texttt{POP}) & = \mathcal{S}.\mathit{remove}(0) \\
(5) & \tau(\mathcal{S}, \texttt{OP}) & = [\texttt{OP}(\mathcal{S}[0], ..., \mathcal{S}[\delta - 1]) \mid \mathcal{S}[\delta : n]\ ]
\end{array}
$$

Fig. 4. Symbolic execution of the instructions that operate on the stack

Once we have the partitioned blocks from the CFG, we aim at obtaining a functional description of the output stack (i.e., the stack after executing the sequence of bytecodes in the block) using symbolic execution for each of the partitioned blocks. As the stack is empty before executing a transaction and the number of elements that each EVM bytecode consumes and produces is known, the size of the stack at the beginning of each block can be inferred statically. We can thus assume that the initial stack size is given within the CFG. A symbolic stack $\mathcal{S}$ is a list of size k that represents the state of the stack where the list position 0 corresponds to the top of the stack and $k - 1$ is the index of the bottom of the stack, such that $\mathcal{S}[i]$ is the symbolic value stored at the position i of the stack. Initially, the input stack maps each index to a symbolic variable s_i.

The symbolic execution of each bytecode is defined using the transfer function τ described in Fig. 4 which takes an input stack and a bytecode and returns the output stack as follows: (1) the `PUSHX` bytecode stores at the top of the stack the value A, (2) `DUPX` duplicates the element stored at position $\texttt{X}{-}1$ to the top of the stack, (3) `SWAPX` exchanges the values stored at the top of the stack with the one stored at position `X`, (4) `POP` deletes the value stored in the top of the stack (using the list operation *remove* to delete the element at the given position), (5) `OP` represents all other EVM bytecodes that operate with the stack (arithmetic and bit-wise operations among others). In that case, τ creates a symbolic expression that is a functor with the same name as the original EVM bytecode and as arguments the symbolic expressions stored in the stack elements that it consumes. Here, δ stands for the number of elements that the EVM bytecode `OP` gets from the stack. Now, the SFS can be defined using the function τ as follows.

Definition 2 (SFS). *Given a block B with an initial size of the stack k, the initial state of the stack $\mathcal{S}_0$ stores at each position $i \in \{0, ..., k-1\}$ a symbolic variable s_i. Then, the transfer function τ is extended to the block B, denoted by $\tau(B)$, as: $[s_0, \ldots, s_{k-1}]$ if B is empty; and $\tau(\tau(B'), o)$ if B has o as last operation and B' is the resulting block without o. The SFS of B is $\mathcal{S}_0 \Longrightarrow \mathcal{S} = \tau(B)$.*

Example 3. Consider the block formed by the EVM bytecode shown in Fig. 1, starting with the bytecode at program point 2 (pp2 for short) and finishing with the bytecode at pp27. Before executing the block symbolically, the initial stack is $\mathcal{S}_0 = [s_0, s_1, s_2, s_3, s_4]$ and $k = 5$. After applying the transfer function τ, we obtain the following results at the next selected program points:

$pp2: \ \tau(\mathcal{S}, \texttt{PUSH1 0X00}) = [0, s_0, s_1, s_2, s_3, s_4]$
$pp3: \ \tau(\mathcal{S}, \texttt{DUP1}) = [0, 0, s_0, s_1, s_2, s_3, s_4]$
$pp5: \ \tau(\mathcal{S}, \texttt{DUP6}) = [s_2, 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
$pp6: \ \tau(\mathcal{S}, \texttt{DUP8}) = [s_3, s_2, 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
$pp7: \ \tau(\mathcal{S}, \texttt{ADD}) = [\texttt{ADD}(s_3, s_2), 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
$pp8: \ \tau(\mathcal{S}, \texttt{SWAP2}) = [0, 0, \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
$pp9: \ \tau(\mathcal{S}, \texttt{POP}) = [0, \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
$pp15: \tau(\mathcal{S}, \texttt{DUP1}) = [\texttt{ADD}(s_1, s_0), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
$pp16: \tau(\mathcal{S}, \texttt{DUP3}) = [\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
$pp17: \tau(\mathcal{S}, \texttt{EXP}) = [\texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0)), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_3, s_2), s_0, s_1, s_2, s_3, s_4]$
$pp27: \tau(\mathcal{S}, \texttt{POP}) = [s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$

Thus, altogether, the output stack of the SFS given by τ for the block in Fig. 1 is $\mathcal{S} = [s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$. For example, we can see that τ updates the stack inserting a 0 in the top of the stack at pp2. At pp8, it swaps the element in the top of the stack ($\texttt{ADD}(s_3, s_2)$) with the element stored at position 2 (0). It generates a symbolic expression to represent the addition at pp7 with the values stored in the position of the stack that it consumes. At pp17 it generates a new symbolic expression $\texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))$ to represent the exponentiation of the two elements stored in the top of the stack. Note that in this case these elements are also symbolic expressions of the two previous additions symbolically executed before.

Finally, we capture optimizations based on the semantics of the arithmetic and bit-wise operations, by applying simplification rules on the SFS of the block before we proceed to generate the optimized code. This simplification besides reducing the number of operations includes other notions of simplification as well. The easiest examples are the application of simplification rules like with the units of every operation, or with the idempotence of bit-wise Boolean operators.

4 Optimal Synthesis Using Max-SMT

This section describes our Max-SMT encoding. We start by preprocessing the SFS into an abstract form that is convenient for the encoding in Sect. 4.1. Next, Sect. 4.2 describes a key element of our encoding: the stack model. Sect. 4.3 presents the complete encoding of the problem and Sect. 4.4 how to obtain the optimized EVM blocks from the model obtained by the SMT solver. Finally, Sect. 4.5 describes the optimization problem. The SFS and the encoding generated for the example shown in Fig. 1 are available at https://github.com/mariaschett/syrup-backend/tree/master/examples/cav2020.

4.1 Abstracting Uninterpreted Functions

Before we apply our encoding, we need to abstract all (sub)expressions occurring in the SFS, by introducing new fresh variables $s_k, s_{k+1}, \ldots$ that start after the last stack variable in the initial stack $[s_0, \ldots, s_{k-1}]$ (of size k). In this process we have a mapping from fresh variables to shallow expressions of depth one, i.e., built with a function symbol and variables or constants as arguments. Here we introduce the *minimal* number of fresh variables that allow us to describe the SFS using only shallow expressions. By minimal, we mean that we use the same variable if some subterm occurs more than once (we also take into account commutativity properties to avoid creating unnecessary fresh variables). Finally if an uninterpreted function occurs more than once, we add a subscript from 0 on to distinguish them. As a result we have that the *abstracted SFS* is defined by a stack S containing only stack variables, fresh variables or constants (in $\{0, \ldots, 2^{256} - 1\}$) and a map M from fresh variables to shallow terms formed by an uninterpreted function (maybe with subscript) applied to stack variables, fresh variables or constants (in $\{0, \ldots, 2^{256} - 1\}$). Besides, we note that the *abstracted SFS* generated is equivalent to first-order A-normal form with shearing. Trivially, all positions in the stack in the SFS and the abstracted SFS are equal when the map is fully applied to remove all fresh variables and the subscripts are removed. Moreover, we have that every uninterpreted function of the SFS has a fresh variable assigned in the map and all function symbols in the map are different.

Example 4. The abstraction of the SFS $[s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$ shown in Example 3 needs three fresh variables s_5, s_6 and s_7. Then, the abstracted SFS is the stack $S = [s_4, s_7]$ and the mapping M is defined as $\{s_5 \mapsto \texttt{ADD}_0(s_3, s_2), s_6 \mapsto \texttt{ADD}_1(s_1, s_0), s_7 \mapsto \texttt{EXP}(s_5, s_6)\}$.

4.2 Modeling the Stack

A key element in our encoding is the representation of the stack and the elements it contains. As mentioned in Sect. 2.3, a first observation is that in our approach we will only have in the stack constants in the domain $\{0, \ldots, 2^{256} - 1\}$ (we do not care if they represent a negative number or not, as they are handled simply as 256-bit words), initial stack variables $s_0, \ldots, s_{k-1}$ and fresh variables $s_k, \ldots, s_v$. In order to distinguish between constants and the variables s_i, we assign to every variable s_i, with $i \in \{0, \ldots, v\}$, the constant $2^{256} + i$. Now, for instance, we can establish that a `PUSH` operation can only introduce a constant in $\{0, \ldots, 2^{256} - 1\}$ and that fresh variables s_i can only be introduced by uninterpreted functions if the appropriate arguments are in the stack (see below). The rest of stack operations, like `DUP` or `SWAP`, just duplicate or move whatever is in the stack. Since in our encoding we will use the variables $s_0, \ldots, s_v$, as they are part of the SFS, we have a first constraint assigning the constant values to all these variables (this could be done as well with a *let* expression).

$$S_V = \bigwedge\nolimits_{0 \leqslant i < v} s_i = 2^{256} + i$$

Let us now show how we model the stack along the execution of the instructions. First, we have to fix a bound on the number of operations b_o and the size of the stack b_s. We can apply different heuristics to this end though considering the initial number of operations and the maximum number of stack elements involved in the block are sound bounds. We have to express a stack of b_s positions after executing j operations with $j \in \{0, \ldots, b_o\}$. To this end, on the one hand, we use existentially quantified variables $x_{i,j} \in \mathbb{Z}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$ to express the word at position i of the stack after executing the first j operations of the code, where $x_{0,j}$ encodes the word on the top of the stack. On the other hand to complete the modeling we introduce propositional variables $u_{i,j}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$, to denote the *utilization* of the stack (i.e., the words that the stack currently holds). Here, $u_{i,j}$ indicates that the word at position i of the stack after executing the first j operations exists or not.

Additionally, to simplify the next definitions we have the following parameterized constraint that, given an instruction step j with $0 < j \leq b_o$, two stack positions α and β and a shift amount $\delta \in \mathbb{Z}$, with $0 \leq \alpha$, $0 \leq \alpha + \delta$, $\beta < b_s$ and $\beta + \delta < b_s$, imposes that the stack after executing $j + 1$ instructions between positions α and β is the same as the stack after executing the j instruction but with a shift of δ (they are moved up if negative and moved down otherwise).

$$Move(j, \alpha, \beta, \delta) = \bigwedge\nolimits_{\alpha \leqslant i \leqslant \beta} u_{i+\delta,j+1} = u_{i,j} \ \wedge \ x_{i+\delta,j+1} = x_{i,j}$$

4.3 Encoding of Instructions

Let $\mathcal{I}$ be the set of instructions occurring in our problem. The set $\mathcal{I}$ is split in three subsets $\mathcal{I}_C \uplus \mathcal{I}_U \uplus \mathcal{I}_S$, where:

- $\mathcal{I}_C$ contains the commutative uninterpreted functions occurring in the map M of the abstracted SFS,
- $\mathcal{I}_U$ contains the non-commutative uninterpreted functions occurring in M,
- $\mathcal{I}_S$ contains the stack operations: `PUSH`, that introduces an up to 32-bytes item on top of the stack; `POP` that removes the top of the stack; `DUP`k, with $k \in \{1, \ldots, 16\}$ that copies the $k-1$ element of the stack on top of the stack; `SWAP`k, with $k \in \{1, \ldots, 16\}$ that swaps the top of the stack with the k element of the stack; and an extra operation `NOP` that does nothing.

Note that, although in EVM there are 32 different `PUSH` instructions depending on the amount of bytes needed to express the item, in our context this distinction is unnecessary, since we can decide afterwards which `PUSH` do we need by checking in the obtained solution which is the value to be pushed. Also, the operations `DUP`k in $\mathcal{I}_S$ are reduced to only those with $k < b_s$ (otherwise we go beyond the maximal size of the stack) and, similarly, the operations `SWAP`k in $\mathcal{I}_S$ are reduced to only those with $k < b_s$.

Let θ be a mapping from the set of instructions in $\mathcal{I}$ to consecutive different non-negative integers in $\{0, \ldots, m_\iota\}$, where $m_\iota + 1$ is the cardinality of $\mathcal{I}$. In order to encode the selected instructions at every step, we introduce the existentially quantified variables $t_j \in \{0, \ldots, m_\iota\}$, with $j \in \{0, \ldots, b_o - 1\}$ where for every instruction $\iota \in \mathcal{I}$, if $t_j = \theta(\iota)$ then we have that the operation executed at step j is ι. Additionally, we introduce associated existentially quantified variables $a_j \in \{0, \ldots, 2^{256} - 1\}$, with $j \in \{0, \ldots, b_o - 1\}$, to express the value pushed at the top of the stack when $t_j = \theta(\texttt{PUSH})$ (otherwise the value of a_j is meaningless).

Encoding the Stack Operations. First we show how we encode the effect of choosing in t_j one of the operations in $\mathcal{I}_S$ that does not depend on the particular (abstracted) SFS we are considering. The following parameterized constraints show this effect:

$$\begin{aligned}
C_{\texttt{PUSH}}(j) &= t_j = \theta(\texttt{PUSH}) \Rightarrow 0 \leq a_j < 2^{256} \wedge \neg u_{b_s-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = a_j \wedge \\
&\qquad \mathit{Move}(j, 0, b_s-2, 1)\\
C_{\texttt{DUP}k}(j) &= t_j = \theta(\texttt{DUP}k) \Rightarrow \neg u_{b_s-1,j} \wedge u_{k-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k-1,j} \wedge \\
&\qquad \mathit{Move}(j, 0, b_s-2, 1)\\
C_{\texttt{SWAP}k}(j) &= t_j = \theta(\texttt{SWAP}k) \Rightarrow u_{k,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k,j} \wedge u_{k,j+1} \wedge \\
&\qquad x_{k,j+1} = x_{0,j} \wedge \mathit{Move}(j, 1, k-1, 0) \wedge \\
&\qquad \mathit{Move}(j, k+1, b_s-1, 0)\\
C_{\texttt{POP}}(j) &= t_j = \theta(\texttt{POP}) \Rightarrow u_{0,j} \wedge \neg u_{b_s-1,j+1} \wedge \mathit{Move}(j, 1, b_s-1, -1)\\
C_{\texttt{NOP}}(j) &= t_j = \theta(\texttt{NOP}) \Rightarrow \mathit{Move}(j, 0, b_s-1, 0)
\end{aligned}$$

Notice that the stack before executing the instruction t_j is given in the variables $x_{0,j}, \ldots, x_{b_s-1,j}$ and $u_{0,j}, \ldots, u_{b_s-1,j}$, while the stack after executing t_j is given in $x_{0,j+1}, \ldots, x_{b_s-1,j+1}$ and $u_{0,j+1}, \ldots, u_{b_s-1,j+1}$.

In order to avoid redundant solutions (with `NOP` in intermediate steps), we have to add as well a constraint stating that once we choose `NOP` as instruction t_j we can only choose `NOP` for the following instructions $t_{j+1}, t_{j+2}\ldots$:

$$C_{\texttt{fromNOP}} = \bigwedge\nolimits_{0 \leqslant j < b_o - 1} \; t_j = \theta(\texttt{NOP}) \Rightarrow t_{j+1} = \theta(\texttt{NOP})$$

Encoding the Uninterpreted Operations. The encoding of the uninterpreted operations comes from the map M of the abstracted SFS. First of all, note that, every function f occurs only once in M (since subscripts are introduced) and for every $r \mapsto f(o_0, \ldots, o_{n-1})$ in M we have that $f \in I_C \uplus \mathcal{I}_U$, r is a fresh variable, and $o_0, \ldots, o_{n-1}$ are either initial stack variables, fresh variables or constants. Note also that if $f \in \mathcal{I}_C$ then $n = 2$. Therefore, we define in the encoding the effect of choosing in t_j the uninterpreted function f with $r \mapsto f(o_0, \ldots, o_{n-1})$ in M, as an operation that takes its arguments $o_0, \ldots, o_{n-1}$ from the stack and places its result r in the stack (where o_0 must be at the top of the stack).

$$\begin{aligned}
C_U(j, f) = \; t_j = \theta(f) \Rightarrow & \bigwedge\nolimits_{0 \leqslant i \leqslant n-1} (u_{i,j} \;\wedge\; x_{i,j} = o_i) \;\wedge u_{0,j+1} \;\wedge\; x_{0,j+1} = r \;\wedge \\
& \mathit{Move}(j, n, \mathit{min}(b_s - 2 + n, b_s - 1), 1 - n) \wedge \\
& \bigwedge\nolimits_{b_s - n + 1 \leqslant i \leqslant b_s - 1} \neg u_{i,j+1} \\
& \text{where } f \in \mathcal{I}_U \text{ and } r \mapsto f(o_0, \ldots, o_{n-1}) \in M
\end{aligned}$$

Now for the commutative functions the only difference is that we know that $n = 2$ and that we can find the arguments in any of both orders in the stack:

$$\begin{aligned}
C_C(j, f) = t_j = \theta(f) \Rightarrow\; & u_{0,j} \wedge u_{1,j} \wedge \\
& ((x_{0,j} = o_0 \wedge x_{1,j} = o_1) \vee (x_{0,j} = o_1 \wedge x_{1,j} = o_0)) \wedge \\
& u_{0,j+1} \wedge x_{0,j+1} = r \wedge \mathit{Move}(j, 2, b_s - 1, -1) \wedge \neg u_{b_s-1,j} \\
& \text{where } f \in \mathcal{I}_C \text{ and } r \mapsto f(o_0, o_1) \in M
\end{aligned}$$

Finding the Target Program. We assign to every $\iota \in \mathcal{I}$ an integer. Then, $t_j \in \mathbb{Z}$ encodes the chosen instruction at position j in the target program for $0 \leqslant j < b_o$. To encode the selection of an instruction for every t_j, we have the following constraint:

$$\begin{aligned} C_{\mathcal{I}} = C_{\texttt{fromNOP}} \ \wedge \ \textstyle\bigwedge_{0\leqslant j<b_o} & \ 0 \le t_j \le m_\iota \ \wedge \\ & C_{\texttt{PUSH}}(j) \ \wedge \ C_{\texttt{DUP}k}(j) \ \wedge \ C_{\texttt{SWAP}k}(j) \ \wedge \ C_{\texttt{POP}}(j) \ \wedge \\ & C_{\texttt{NOP}}(j) \ \wedge \ \textstyle\bigwedge_{f\in\mathcal{I}_U} C_U(j,f) \ \wedge \ \textstyle\bigwedge_{f\in\mathcal{I}_C} C_C(j,f)) \end{aligned}$$

Complete Encoding. Let us conclude our encoding by defining the formula C_{SFS} that states the whole problem of finding an EVM block for a given initial stack $[s_0,\ldots,s_{k-1}]$ and abstracted SFS with final stack $[f_0,\ldots,f_{w-1}]$ and map M. Hence, we introduce a constraint B to describe how the stack at the beginning is and a constraint E to describe how the stack at the end is and combine all the constraints defined above to express C_{SFS}.

$$\begin{aligned} B \quad &= \textstyle\bigwedge_{0\leqslant\alpha<k}(u_{\alpha,0} \ \wedge \ x_{\alpha,0} = s_\alpha) \ \wedge \ \textstyle\bigwedge_{k\leqslant\beta\leqslant b_s-1} \neg u_{\beta,0} \\ E \quad &= \textstyle\bigwedge_{0\leqslant\alpha<w}(u_{\alpha,b_o} \ \wedge \ x_{\alpha,b_o} = f_\alpha) \ \wedge \ \textstyle\bigwedge_{w\leqslant\beta\leqslant b_s-1} \neg u_{\beta,b_o} \\ C_{SFS} &= S_V \ \wedge \ C_{\mathcal{I}} \ \wedge \ B \ \wedge \ E \end{aligned}$$

Finally, let us mention that the performance of the used SMT solvers greatly improves when the following (redundant) constraint, which states that all functions in $\mathcal{I}_U \uplus \mathcal{I}_C$ should be eventually used, is added: $\bigwedge_{\iota\in\mathcal{I}_U\uplus\mathcal{I}_C} \bigvee_{0\leqslant j<b_o} t_j = \theta(\iota)$

Empirical evidence shows, that this constraint helps the solver to establish optimality, and removing it increases the time-outs and time taken by roughly 50%. On the other hand, adding the similar constraint that all functions in $\mathcal{I}_U \uplus \mathcal{I}_C$ are used at most once, while also helping the solvers to show optimality for already optimal blocks, the performance for finding optimizations decreases by a similar rate. As the latter is our main motivation, we did not include the constraint.

4.4 From Models to EVM Blocks

The following definition shows how we can extract a concrete set of operations from a model for the formula C_{SFS} that computes the given SFS.

Definition 3. *Given a model σ for C_{SFS} we have that block(σ) is defined as the sequence of EVM operations $o_0,\ldots,o_f$ where f is the largest $j \in \{0,\ldots,b_o-1\}$ such that $t_j \neq \theta($NOP$)$. Now for all $\alpha \in \{0,\ldots,f\}$ the operation o_α is taken as*

1. *$o_\alpha =$ PUSHk a_α if $t_\alpha = \theta($PUSH$)$ and a_α can be represented with k bytes.*
2. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota)$ where $\iota \in \mathcal{I}_S \setminus \{$PUSH$\}$*
3. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota)$ where $\iota \in \mathcal{I}_U \uplus \mathcal{I}_C$ and ι has no subscript.*
4. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota_l)$ where $\iota_l \in \mathcal{I}_U \uplus \mathcal{I}_C$ and has subscript l.*

The following result easily follows from the construction of C_{SFS}.

Theorem 1 (soundness). *Given an SFS and values for b_o and b_s, we have that if σ is a model for C_{SFS} obtained from the abstracted SFS then block(σ) computes the given SFS.*

4.5 Optimization Using Max-SMT

Now that we know that every model of C_{SFS} provides a block that computes the SFS, we want to obtain the optimal solution. Since the cost of the solution can be expressed in terms of the cost of every of the instructions we select in all t_j, we will introduce soft constraints expressing the cost of every selection. A (partial weighted) Max-SMT problem is an optimization problem where we have an SMT formula which establishes the *hard constraints* of the problem and a set of pairs $\{[C_1, \omega_1], \ldots, [C_m, \omega_m]\}$, where each C_i is an SMT clause and ω_i is its weight, that establishes the *soft constraints*. In this context, the optimization problem consists in finding the model that satisfies the hard constraints and minimizes the sum of the weights of the falsified soft constraints. Our approach to find the optimal code is by encoding the problem as a Max-SMT optimization problem, where we add to the SMT formula C_{SFS} which defines our *hard constraints* a set of *soft constraints* such that sum of the weights of the falsified soft constraints coincides with the cost (in terms of gas) of the operations taken in every step. Therefore the optimal solution to the Max-SMT problem coincides with the optimal solution in terms of gas cost.

In the EVM, every operation has an associated gas cost, which in general is constant, but in some few cases may depend on the particular arguments it is applied to or on the state of the blockchain. All these operations that are non-constant are considered as uninterpreted, and hence we cannot change the operands on which they are applied. Therefore, omitting the non-constant part cannot affect which is the optimal solution. Thanks to this, we can split our set of instructions $\mathcal{I}$ in $p+1$ disjoint sets $W_0 \uplus \ldots \uplus W_p$ where all instructions in W_i have the same constant cost cost_i, and such that the costs are strictly increasing, i.e., $\mathsf{cost}_0 = 0$ and $\mathsf{cost}_{i-1} < \mathsf{cost}_i$ for all $i \in \{1, \ldots, p\}$.

In the following we describe the encoding we have chosen for the weighted clauses (we have tried other slightly simpler alternatives but, in general, they behave worse). Let $w_i = \mathsf{cost}_i - \mathsf{cost}_{i-1}$ for $i \in \{1, \ldots, p\}$. Hence, we have that $w_i > 0$ and, moreover, $\mathsf{cost}_i = \Sigma_{1 \leqslant \alpha \leqslant i} w_\alpha$ for $i \in \{1, \ldots, p\}$. Then, our Max-SMT problem O_{SFS} is obtained adding to C_{SFS} the following soft constraints

$$O_{SFS} = C_{SFS} \ \wedge \bigwedge_{0 \leqslant j < b_o} \ \bigwedge_{1 \leqslant i \leqslant p} \ [\bigvee_{\iota \in W_0 \uplus \ldots \uplus W_{i-1}} t_j = \theta(\iota) \ , \ w_i \ ]$$

Therefore, if the selected instruction at step j is ι (i.e. $t_j = \theta(\iota)$) for some $\iota \in W_i$ then we accumulate the weight w_α of all soft clauses with $\alpha \in \{1, \ldots, i\}$, which as said sums cost_i, and hence we accumulate the cost of executing the instruction ι. From this fact, our optimality theorem follows.

Theorem 2 (optimality). *Given an SFS P and values for b_o and b_s, we have that if σ is the optimal solution for the weighted Max-SMT problem O_{SFS} obtained from the abstracted SFS of P, then block(σ) is the optimal code that has an SFS equal to P modulo commutativity.*

5 Experimental Evaluation

This section presents the results of our evaluation using syrup, the SYnthesizeR of sUPer-optimized smart contracts that implements our approach. Our tool syrup uses EthIR [6] to generate the CFGs of the analyzed contracts and Z3 [12] version 4.8.7, Barcelogic [7], and MathSAT [11] version 1.6.3 (namely its optimality framework OptiMathSAT), as SMT solvers. We refer by s-Z3, s-Bar, s-OMS, to the results of using syrup with the respective solvers. Experiments have been performed on a cluster with Intel Xeon Gold 6126 CPUs at 2.60 GHz, 2 GB of memory and timeout of 15 min, running CentOS Linux 7.6. The main components of syrup are implemented in Python and OCaml. The backend of syrup generating SMT constraints from a SFS is open-source and can be found at github.com/mariaschett/syrup-backend. Our tool accepts smart contracts written in versions of Solidity up to 0.4.25 and EVM bytecode v1.8.18, namely the three new EVM bytecodes (`SHL`, `SHR` and `SAR`) introduced from the Solidity compiler version 0.5.0 are not handled yet by EthIR. Our experimental setup consists of two groups of benchmarks:

(i) In order to compare with the existing tool ebso, we use the same data set (and the results for ebso) from [20]: the blocks of the 2500 most called contracts deployed on the Ethereum blockchain[3] after removing the duplicates and the blocks which are only different in the arguments of `PUSH` by abstracting to word size 4 bit. This results in a data set of 61 217 blocks.

(ii) A more realistic setting in which we analyze the 150 most called contracts[4] queried from the Ethereum blockchain and removing those of the versions not supported, resulting in 128. As the dates in which the contracts are fetched are different, not all 128 contracts are included in setup (i), indeed, the intersection are 106 contracts (besides there might be updated versions). This setting is more realistic since the analysis is performed at the contract-level (without removing any duplicates or similar blocks) and allows us to gather statistics to assess the gains at the level of the deployed contracts.

We note that analyzing the most called contracts corresponds to the most relevant case study as, according to [16], many Ethereum contracts are not used.

5.1 Comparison with ebso (setup I)

As seen in Definition. 1, we split the 61 217 blocks on certain bytecodes that are not optimized, leading to a total of 72 450. For comparison, we merge the split blocks back together. The next table shows the results of optimizing the 61 217 blocks by ebso (first column), and by syrup for every solver (next columns). In column s-All, we use the 3 solvers as a single framework in syrup that yields the best solution returned by any of the solvers (in parenthesis we show percentages).

[3] Up to Ethereum blockchain block number 7 300 000 until 2019-03-04 01:22:15 UTC.
[4] Up to Ethereum blockchain block number 9 193 265 until 2019-12-31 23:59:45 UTC.

	ebso	s-Z3	s-Bar	s-OMS	s-All
A	3882 (6.34%)	20 636 (33.71%)	20 783 (33.95%)	20 973 (34.26%)	20 988 (34.28%)
O	393 (0.64%)	25 922 (42.34%)	26 458 (43.22%)	28 063 (45.84%)	28 195 (46.06%)
B	550 (0.90%)	6288 (10.27%)	3051 (4.98%)	5293 (8.65%)	5726 (9.35%)
N	n/a	1933 (3.16%)	563 (0.92%)	837 (1.37%)	1020 (1.67%)
T	56 392 (92.12%)	6438 (10.52%)	10 362 (16.93%)	6051 (9.88%)	5288 (8.64%)
G	27 726	1 188 311	1 003 717	1 272 381	1 309 875
S	Not avail	13 710 904.75	13 141 046.21	12 239 980.85	10 948 011.57

Row **A** shows the number of blocks that were *A*lready optimal, i.e., those that cannot be optimized because they already consume the minimal amount of gas and ebso/syrup find bytecode with the same consumption. Row **O** contains the number of blocks that have been optimized and the found solution has been proven to be *O*ptimal, i.e., the one that consumes the minimum amount of gas needed to obtain the SFS provided. The solvers used are able to provide the best solution found until the timeout is reached. Row **B** contains the number of blocks that have been optimized into a *B*etter solution that consumes less gas but it is not shown to be the optimum. Row **N** shows the number of blocks that have *N*ot been optimized and not proven to be optimal, i.e., the solution found is the original one but there may exist a better one. Row **T** contains the number of blocks for which no model could be found when the *T*imeout was reached. Row **G** contains the accumulated *G*as savings for all optimized blocks. Importantly, the real savings would be larger if the optimized blocks are part of a loop and hence might be executed multiple times. Row **S** shows the time in *S*econds in which each setting analyzes all the blocks.

Let us first compare the results by ebso and our best results when using the portfolio of solvers in s-All. It is clear from the figures that syrup significantly outperforms ebso on the number of blocks handled (while ebso times out in 92.12 % of the blocks, we only timeout in 8.64 %) and on the overall gas gains (two orders of magnitude larger). For the analyzed blocks (i.e., those that do not timeout), the percentages of syrup for number of optimized into better blocks, into optimal blocks, and those proven to be already optimal, are much larger than those of ebso. We now discuss how the gains for the blocks that ebso can analyze compare to the gains by syrup. In particular, if missing part of the semantics of the uninterpreted instructions and the `SSTORE` bytecode significantly affects the gains. Out of 943 examples, where ebso found an optimization, in 46 cases syrup proved optimality w.r.t. the SFS and saved 348 gas but saved less gas than ebso (total 10 514 gas). The source of this gain is the `SSTORE` bytecode: there are two blocks where ebso saves 5000 each, because it realizes that we read from a key in storage to then store the value back unchanged. As we discuss in Sect. 7, our framework naturally extends to handle this storage optimization. However, in nearly all of 393 cases, where ebso found an optimal solution—in 378 cases—syrup saves as much as ebso amounting to 2670 gas. That is, the additional semantics did not improve savings. Furthermore, in 43 cases out of 943, the semantics did impede ebso's performance so that syrup found a better result with 597 gas versus 440 of ebso. Therefore, we can conclude that syrup is far more scalable and precise than ebso, the cases in which syrup optimizes less than ebso are seldom and can be naturally handled in the future. Moreover, they are offset by the cases where syrup did find an optimization, whereas ebso did not.

Finally, we can see that MathSAT is the solver that shows the best performance: It proves optimality of 34.26 % and optimizes 54.49 % of the blocks (c.f. Sect. 5.3). Regarding analysis time, the global figure is not reported in [20]. In syrup, by accumulating the time of all four scenarios (s-X) and using the 900 s timeout of ebso, we analyze the whole data set in about 3042 h. We note that, by considering the solvers as a portfolio, we reduce the analysis time as when an optimal solution is found, the execution of the other two solvers is stopped. However, for the other cases, we take the highest time taken by the solvers as we need to know all solutions in order to keep the best one and provide an answer.

5.2 Analysis of the Most Called Contracts with Gas Savings (setup Ii)

For our second setup, syrup produces the following results for the 46 966 blocks of the 128 (most called) smart contracts:

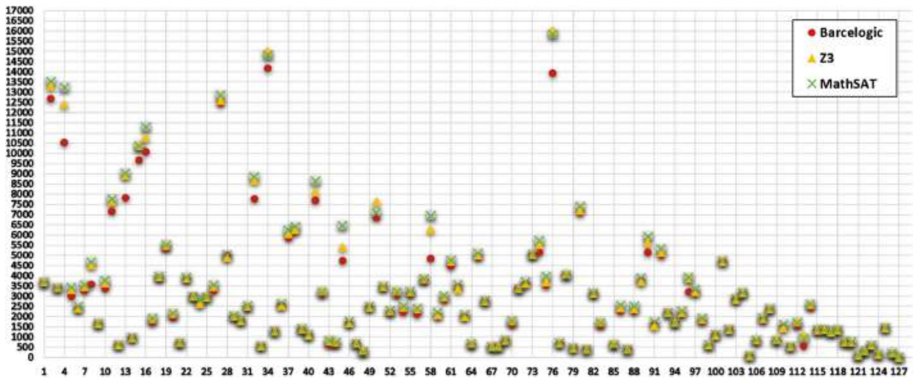


Fig. 5. Gas saved per contract in the 128 most called smart contracts

	s-Z3	s-Bar	s-OMS	s-All
A	30 846 (65.68%)	30 923 (65.84%)	30 971 (65.94%)	30 974 (65.95%)
O	13 102 (27.9%)	13 240 (28.19%)	13 586 (28.93%)	13 606 (28.97%)
B	933 (1.98%)	510 (1.09%)	746 (1.59%)	801 (1.71%)
N	695 (1.48%)	95 (0.2%)	295 (0.63%)	467 (0.99%)
T	1390 (2.96%)	2198 (4.68%)	1368 (2.91%)	1118 (2.38%)
G	438 483	406 086	437 165	443 248
S	2 919 830.35	2 682 469.58	2 413 612.39	2 378 446.26

As before, MathSAT is the solver that shows the best performance: It proves optimality of 65.94% and optimizes 30.52% of the blocks. The overall gas savings in **G** amount to 0.73% of the total gas which, assuming a uniform distribution of this saving among the contracts, amounts to around a million dollars from 2017 to 2019 (see Sect. 1 for details on this estimations). Moreover, we have calculated that the 64% of the saved gas is due to the simplification rules and the 36% to the Max-SMT optimization, which shows that both parts are highly relevant in our results. For this data set, we additionally display in Fig. 5 the amount of gas saved for each contract. The X-axis corresponds to each of the 128 analyzed contracts and the Y-axis corresponds to the amount of gas saved when using each solver. In general the gains obtained by the different solvers are quite aligned. On average, each contract saves 3425.65 units of gas using Z3, 3172.55 using Barcelogic and 3415.35 using MathSAT. However, we can observe that the gains are dispersed w.r.t. the mean, and there are big differences in the savings obtained for each of the contracts (the standard deviation is 2798.19 for Z3, 2664.05 for Barcelogic and 2889.01 for MathSAT). The biggest amount of gas optimized in all contracts is 18 989 gas using Z3, 18 704 using Barcelogic and 19 205 using MathSAT. In the case of this contract, MathSAT optimizes 706 blocks out of 1910, and the highest amount of gas optimized is 162 though the most common amount of gas optimized is 3 (in 165 blocks). The highest amount of gas optimized per block in all contracts is 481. Finally, we have analyzed the impact of our optimization on the function `transfer` of the AirdropToken smart contract, that has been called around 520 000 times. For this function, which has no loops, syrup saves 832 units of gas per call. From the number of calls per day (obtained from [2]), we estimate a total saving (just for this function) of 2815 $.

5.3 Comparison of SMT Solvers in Precision and Time

Figure 6 aims at providing some data to compare the accuracy and efficiency of the process using the three SMT solvers. The table to the left shows in: **Unique** the number of blocks that are uniquely optimized by the corresponding solver, in **UOptim** the number of blocks that are proven to be optimal uniquely by one solver, and **+GSave** the number of blocks for which one solver has strictly more gains that the others. The suffixes 1 and 2 refer to the data set in Sects. 5.1 and

	s-Z3	s-Bar	s-OMS
Unique1	608	73	925
UOPtim1	22	108	1296
+GSave1	694	634	4286
Unique2	238	6	234
UOPtim2	6	14	237
+GSave2	107	79	563

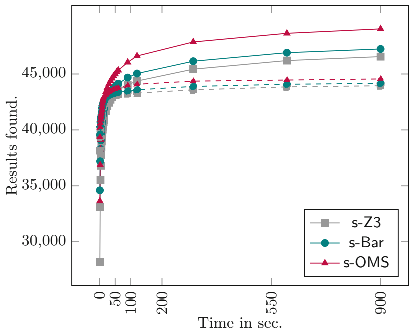


Fig. 6. Comparison of SMT Solvers

5.2, resp., excluding all timeouts. In both data sets, MathSAT uniquely finds a result, uniquely shows the block optimal, or finds the best gain for the large majority. But clearly, in both data sets, every solver was needed to get the best possible solution in every category. The plot to the right of Fig. 6 displays the amount of blocks (Y-axis) that are solved in the corresponding amount of time (X-axis). Dashed lines correspond to data set 1 and plain lines to 2. We can see that for data set (i) within 10 s, nearly 89% of the results were found. For data set (ii) this is even more pronounced, after 10 s around 95% were found, with around 90% already being available after 1 s. The analysis shows that most results can be found very fast and thus our optimizer could be invoked during the compilation of a smart contract without adding a large overhead to compilation.

6 Related Work

There are currently two automated approaches to gas optimization of Ethereum smart contracts. (i) First, the closest to ours is blockchain superoptimization [20], whose goal is the same as ours: find the gas-optimal block of code for each of the blocks in the CFG of the smart contract. While the approach of [20] would not be applicable within a compiler (e.g., it times out in 92.12 % of the blocks used in their experimental evaluation), our optimization tool performs very efficiently (e.g., we have seen that 89% of the blocks are optimized in less than 10 s). The reasons for our efficiency are indeed the fundamental differences with [20]: (1) we use the SFS to solve the optimization problem efficiently as a synthesis problem in which the semantic optimizations are carried out within the SFS part, (2) we do not encode the semantics of the arithmetic and bit-vector operations in the SMT problem, as [20] does, what allows us to express the problem using only existential quantification, (3) we use Max-SMT to solve the optimization problem. The basis for ebso is in [15], where the description of an encoding of unbounded superoptimization with the idea to shift the search for optimal program to the SMT solver is first found. (ii) Second, the system GASOL [5] incorporates also an automatic optimization for storage operations that consists in replacing accesses to the storage (i.e., bytecodes `SSTORE` and `SLOAD`) by equivalent accesses to memory locations (i.e., bytecodes MSTORE and MLOAD), when a static analysis identifies that it is sound and efficient doing such transformation. This optimization is not intra-block, as done in supercompilation, therefore it is not achievable by our approach as it involves modifying multiple blocks, and also requires an analysis that identifies the patterns and the soundness of the transformation. On the other hand, Gasol is not able to make the intra-block optimizations that we are achieving. Therefore, the optimizations in Gasol are orthogonal (and complementary) to those achievable by means of superoptimization.

There is work also focused on identifying gas expensive patterns: (1) the work in [9] identifies 7 expensive patterns on Solidity contracts and proposes optimizations for them. However, there is no tool in [9] that carries out these optimizations automatically; (2) The work in [10] identifies 24 anti-patterns, e.g. [OP,POP] optimizes to POP. Again, there is not automation and those patterns are manually identified. There is recent work that experimentally proves that the gas model for some EVM instructions is not correctly aligned with respect to the observed computational costs in real experiments [26], and that these misalignments can lead to gas-related attacks [22]. Our work is parametric on the gas model used, and new adjustments in the gas model of Ethereum are integrated in our optimizer by just updating the cost for the corresponding modified instructions in our implementation. Finally, the tool TOAST [8] also superoptimizes machine code. Although applied to different settings, the performance of syrup is significantly better and the notions of optimality used are different (sequence length and gas-usage respectively).

7 Conclusions and Future Work

We have presented a novel method for gas super-optimization of smart contracts that combines symbolic execution with an effective Max-SMT encoding. Our focus is on the stack operations because these bytecode operations allow for multiple reorderings, simplifications, and cover the major part of the potential optimizations; while reading and/or writing on memory or storage can be seldom optimized (unless the same value is written, or read, consecutively). In spite of this, the same methodology we have formalized for the stack could be extended to optimize the memory and storage bytecode operations. Basically, the symbolic execution phase would extract a functional specification also for memory and for storage that would be analogous to our SFS and that could include storage-related optimizations (e.g., detecting unnecessary storage). The SMT encoding for these operations would be similar to ours but, for soundness, would have to maintain the order among the memory and storage accesses. It is part of our future work to implement also the super-optimizations for memory and storage and experimentally evaluate if there is significant gain. We also plan to extend the SMT encoding to include information gained from the original program such as the original cost. Currently, in roughly 0.05% of the blocks of Sect. 5.2, syrup synthesizes a more expensive solution.

References

1. The Michelson Language. https://www.michelson-lang.com
2. Bloxy (2018). https://bloxy.info
3. Etherscan (2018). https://etherscan.io
4. Rattle - An EVM Binary Static Analysis Framework (2018). https://github.com/crytic/rattle
5. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: GASOL: gas analysis and optimization for ethereum smart contracts. TACAS 2020. LNCS, vol. 12079, pp. 118–125. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_7
6. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: a framework for high-level analysis of ethereum bytecode. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 513–520. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_30
7. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_27
8. Brain, M., Crick, T., De Vos, M., Fitch, J.: TOAST: applying answer set programming to superoptimisation. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 270–284. Springer, Heidelberg (2006). https://doi.org/10.1007/11799573_21
9. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In SANER, pp. 442–446. IEEE Computer Society (2017)
10. Chen, T., et al.: Towards saving money in using smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 81–84 (2018)
11. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Ethereum. Solidity (2018). https://solidity.readthedocs.io
14. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. PACMPL **2**(OOPSLA), 116:1–116:27 (2018)
15. Jangda, A., Yorsh, G.: Unbounded superoptimization. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, 23–27 October 2017, pp. 78–88 (2017)
16. Kiffer, L., Levin, D., Mislove, A.: Analyzing ethereum's contract topology. In: Proceedings of the Internet Measurement Conference 2018, IMC 2018, pp. 494–499 (2018)
17. Massalin, H.: Superoptimizer - a look at the smallest program. In: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), pp. 122–126 (1987)

18. Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: The 9th annual HITB Security Conference (2018)
19. Mukhopadhyay, M.: Ethereum Smart Contract Development. Packt publishing, Birmingham (2018)
20. Mesnard, F., Stuckey, P.J. (eds.): LOPSTR 2018. LNCS, vol. 11408. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-13838-7
21. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
22. Pérez, D., Livshits, B.: Broken metre: attacking resource metering in EVM (2019). CoRR, abs/1909.07220
23. Dill, D.L., et al.: Move: A language with programmablere sources. Technical report (2019). https://developers.libra.org/docs/state-machine-replication-paper
24. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. In: 34th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2019) (2019)
25. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2019)
26. Yang, R., Murray, T., Rimba, P., Parampalli, U.: Empirically analyzing ethereum's gas mechanism. In 2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, 17–19 June 2019, pp. 310–319 (2019)

Verification of Quantitative Hyperproperties Using Trace Enumeration Relations

Shubham Sahai[1(✉)], Pramod Subramanyan[1], and Rohit Sinha[2]

[1] Indian Institute of Technology, Kanpur, India
{ssahai,spramod}@cse.iitk.ac.in
[2] Visa Research, Palo Alto, USA

Abstract. Many important cryptographic primitives offer probabilistic guarantees of security that can be specified as quantitative hyperproperties; these are specifications that stipulate the existence of a certain number of traces in the system satisfying certain constraints. Verification of such hyperproperties is extremely challenging because they involve simultaneous reasoning about an unbounded number of different traces. In this paper, we introduce a technique for verifying quantitative hyperproperties based on the notion of trace enumeration relations. These relations allow us to reduce the problem of trace-counting into one of model-counting of formulas in first-order logic. We also introduce a set of inference rules for machine-checked reasoning about the number of satisfying solutions to first-order formulas (aka model counting). Putting these two components together enables semi-automated verification of quantitative hyperproperties on infinite-state systems. We use our methodology to prove confidentiality of access patterns in Path ORAMs of unbounded size, soundness of a simple interactive zero-knowledge proof protocol as well as other applications of quantitative hyperproperties studied in past work.

1 Introduction

Recent years have seen significant progress in automated and semi-automated techniques for the verification of security requirements of computer systems [4, 10,16,19,30,47,50,55]. Much of this progress has built on the theory of *hyperproperties* [21], and these have been used extensively in analysis of whether systems satisfy secure information flow properties [1,2,6,8,15,28,35,37,39,49,57] such as observational determinism [41,55] and non-interference [32]. Unfortunately, the security specification of several important security primitives cannot be captured by secure information flow properties like observational determinism. In particular, observational determinism and non-interference are not applicable when reasoning about algorithms that offer probabilistic – as opposed to deterministic – guarantees of confidentiality and integrity. Prominent examples of security primitives offering probabilistic guarantees include Path ORAM [48] and various zero-knowledge proof protocols.

A promising direction for the verification of such protocols are the class of quantitative hyperproperties [29], one example of which is deniability [12,14]. Deniability states that for every infinitely-long sequence of observations that an adversary makes, there are (exponentially) many different secrets that could have resulted in exactly these observations. Therefore, the adversary learns very little about the secrets in an execution from a particular sequence of observations.

How does one prove a quantitative hyperproperty like deniability? Suppose our goal is to show that for every trace of adversary observations, there exist 2^n traces with the same observations but different secrets. Here n is a parameter of the system, e.g., the length of a password in bits. One option, first suggested by Yasuoka and Terauchi [54] and recently revisited by Finkbeiner, Hahn, and Torfah [29], is to consider the following k-trace property, where $k = 2^n + 1$.

$$\forall \pi_0.\ \exists \pi_1, \pi_2, \ldots, \pi_{2^n}.$$
$$\Big(\bigwedge_{j=1}^{2^n} \mathit{obs}(\pi_0) = \mathit{obs}(\pi_j)\Big) \wedge \Big(\bigwedge_{j=1}^{2^n}\bigwedge_{k=1}^{2^n} (j \neq k) \Rightarrow \mathit{secret}(\pi_j) \neq \mathit{secret}(\pi_k)\Big)$$

The property states that for every trace of the system, there must exist 2^n other traces with identical observations and pairwise different secrets. In the above, $\pi_0, \pi_1, \ldots$ represent trace variables, $\mathit{obs}(\pi_j)$ refers to the trace of adversary observations projected from the trace π_j, while $\mathit{secret}(\pi_j)$ refers to the trace of secret values in the trace π_j. There are at least three problems with the verification of the above property. First, the size of this property grows exponentially with n; verification needs to reason about 2^n traces simultaneously and is not scalable. The second problem is quantifier alternation. Even if we could somehow reason about 2^n traces, we have to show that *for every* trace π_0, there *exist* 2^n other traces satisfying the above condition. The third problem is that the above technique does not work for *symbolic* bounds. While it is possible – at least in principle – to use the above construction by picking a specific value of n, say 16, to show that 2^{16} traces exist that satisfy deniability, we would like to show that the property holds for all n, where n is a state variable or parameter of the transition system. Capturing the dependence of the trace-count bound on parameters, such as n, is important because it shows that the attacker has to work exponentially harder as n increases. Such general proofs are not possible by reduction to a k-trace property because the construction requires k be bounded.

Recent work by Finkbeiner, Hahn, and Torfah [29] has made significant progress in addressing the first two problems by showing a reduction from k-trace property checking into the problem of maximum model counting [31]. However, their technique still produces a propositional formula whose size grows exponentially in the size of the quantitative hyperproperty. Further, model counting itself is a computationally hard problem that is known to be $\#P$-complete, and maximum model counting is even harder. As a result, their technique does not scale well and times out on the verification of an 8-bit leakage bound for an 8-bit password. Finally, their method does not support symbolic bounds, and therefore cannot be used to verify parametric systems; we verify several examples of such systems in this paper (e.g., Path ORAM [48] of symbolic size).

In this work, we propose a new technique for quantitative hyperproperty verification that addresses each of the above problems. Our approach is based on the following insights. First, instead of trying to count the number of traces that have the same observations and different inputs, we instead show injectivity/surjectivity from satisfying assignments of a first-order formula to traces of a transition system. This allows us to bound the number of traces satisfying the quantitative hyperproperty by the number of satisfying solutions to this formula. We introduce the notion of a trace enumeration relation to formalize this relation between the first-order formula and traces of the transition system. An important advantage of the above reduction is that proving the validity of a trace enumeration relation is only a hyperproperty – not a quantitative hyperproperty.

Next, we develop a novel technique to bound the number of satisfiable solutions to a first-order logic formula, which is of independent interest. While this is a hard problem, we exploit the fact that our formulas have a significant amount of structure. We introduce a set of inference rules inspired by ideas from enumerative combinatorics [13,52,56]. These rules allow us to bound the number of satisfying assignments to a formula by making only satisfiability queries.

In summary, our techniques can prove quantitative hyperproperties with symbolic bounds on parametric infinite-state systems. We demonstrate their utility by verifying representative quantitative hyperproperties of diverse applications.

Contributions

1. We introduce a specification language for quantitative hyperproperties (QHPs) over symbolic transition systems and define formal satisfaction semantics for this language. Our specification language is more expressive than past work on QHP specification because it allows the bound to be a first-order formula over the state variables of the transition system.
2. We provide several examples of QHPs relevant to security verification. We identify a new class of QHPs, referred to as soundness hyperproperties, applicable to protocols that provide statistical guarantees of integrity.
3. We propose a novel semi-automated verification methodology for proving that a system satisfies a QHP. Our methodology applies to properties that involve a single instance of quantifier alternation and works by reducing the problem of QHP verification to that of checking non-quantitative hyperproperties over two and three traces of the system and counting satisfiable solutions to a formula in first-order logic.
4. We introduce a set of inference rules for bounding the number of satisfiable solutions to a first-order logic formula, using only satisfiability queries.
5. We demonstrate the applicability of our specification language and verification methodology by providing proofs of security for Path ORAM, soundness of a simple zero-knowledge protocol, as well as examples taken from prior work on quantitative security specifications. We show that our verification

methodology scales to larger systems than could be handled in prior work. To the best of our knowledge, our work is the first machine-checked proof of confidentiality of the access patterns in Path ORAM.

2 Motivating Example

In this section, we first introduce the model of transition systems used in this paper. We then discuss quantitative hyperproperty (QHP) specification and verification for our running example – a simple zero-knowledge puzzle.

2.1 Preliminaries

Let $FOL(\mathcal{T})$ denote first-order logic modulo a theory $\mathcal{T}$. The theory $\mathcal{T}$ is assumed to be multi-sorted, includes the theory of linear integer arithmetic (LIA), and contains the $=$ relation. Let $\Sigma_{\mathcal{T}}$ be the theory $\mathcal{T}$'s signature: the set consisting of the constant, function, and predicate symbols in the theory. We say that a formula is a $\Sigma_{\mathcal{T}}$-formula if it consists of the symbols in $\Sigma_{\mathcal{T}}$ along with variables, logical connectives, and quantifiers. We only consider theories which are such that the set of satisfying assignments for any $\Sigma_{\mathcal{T}}$-formula is a countable set.[1]

For every variable x, we will assume there exists a unique variable x', which we refer to as the primed version of x. We will use X, Y, and Z to denote sets of variables. Given a set of variables X, we will use X' to refer to the set consisting of the primed version of each variable in X, that is $X' = \{x' \mid x \in X\}$. Similarly X_1, X_2, etc. are sets consisting of new variables defined as follows: $X_1 = \{x_1 \mid x \in X\}$ and $X_2 = \{x_2 \mid x \in X\}$. We will use $F(X)$ to denote the application of a function or predicate symbol F on the variables in the set X. A satisfying assignment σ to the formula $F(X)$ is written as $\sigma \models F(X)$. Given a formula $F(X)$ and a satisfying assignment σ to this formula, we will denote the valuation of the variable $x \in X$ in the assignment σ as $\sigma(x)$. We will abuse notation in two ways and also write $\sigma(X)$ to refer to a map from the variables $x \in X$ to their assignments in σ. We will also write $\sigma(G(X))$ to denote the valuation of the term $G(X)$ under the assignment σ.

The number of satisfiable assignments for the variables in the set X to a formula $F(X, Y)$ as a function of the variables Y will be denoted by $\#X.\, F(X, Y)$. $\#X.\, F(X, Y)$ is the function $\lambda Y\ .\ |\{\sigma(X) \mid \sigma \models F(X, Y)\}|$ evaluated at Y; $|S|$ is the cardinality of the set S. For example, consider the predicate $f(i, n) \doteq (0 \leq i < 2n)$. In this case, $\#i.\, f(i, n) = \max(0, 2n)$, meaning that for a given value of $n > 0$, there are $2n$ satisfying assignments to i.

Definition 1 (Transition System). *A transition system M is defined as the tuple $M = \langle X, Init(X), Tx(X, X')\rangle$. X is a finite set of (uninterpreted) constants that represents the state variables of the transition system. Init and Tx are $\Sigma_{\mathcal{T}}$-formulas representing the initial states and the transition relation, respectively.*

[1] Our experiments mostly use the AUFLIA theory which allows arrays, uninterpreted functions, and linear integer arithmetic.

Init is defined over the signature $\Sigma_{\mathcal{T}} \cup X$. Tx is over the signature $\Sigma_{\mathcal{T}} \cup X \cup X'$; X represents the pre-state of the transition and X' represents its post-state.

A state of the system is an assignment to the variables in X. We use $\sigma^0, \sigma^1, \sigma^2$ etc. to represent states. A trace of the system M is an infinite sequence of states $\tau = \sigma^0\sigma^1\sigma^2 \dots\ \sigma^i \dots$ such that $\mathit{Init}(\sigma^0)$ is valid and for all $i \geq 0$, $\mathit{Tx}(\sigma^i, \sigma^{i+1})$ is valid; in order to keep notation uncluttered, we will often drop the ≥ 0 qualifier when referring to trace indices. We assume that every state of the transition system has a successor: for all σ there exists some σ' such that $\mathit{Tx}(\sigma, \sigma')$ is valid, ensuring every run of the system is infinite. We will represent traces by τ, τ_1, τ_2, etc. Given a trace τ, we refer to its i^{th} element by τ^i. If $\tau = \sigma^0\sigma^1 \dots$, then $\tau^0 = \sigma^0$ and $\tau^1 = \sigma^1$. The notation $\tau^{[i,\infty]}$ refers to the suffix of trace τ starting at index i. The set of all traces of the system M is denoted by Φ_M. Given a state σ and a variable $x \in X$, $\sigma(x)$ is the valuation of x in the state σ.

2.2 Motivating Example: Zero-Knowledge Hats

Zero-knowledge (Z-K) proofs are constructions involving two parties: *a prover* and *a verifier*, where the prover's goal is to convince the verifier about the veracity of a given statement without revealing any additional information. We motivate the need for quantitative hyperproperty verification using a Z-K puzzle.

Puzzle Overview: Consider the following scenario. Peggy has a pair of otherwise identical hats of different colors (say, yellow and green). She wants to convince Victor, who is yellow-green color blind, that the hats are of different colors, without revealing the colors of the hats. This problem can be solved using the following interactive protocol. Peggy gives both hats to Victor, and Victor randomly chooses a hat behind a curtain and shows it to Peggy. Next, he goes back behind the curtain and uniformly randomly chooses if he wants to switch the hat or not. He now appears in front of Peggy and asks: "Did I switch?"

If the hats are really of different colors, Peggy will be able to answer correctly with probability 1. If Peggy is cheating – the hats are in fact of the same color – her best strategy is to guess, and with probability 0.5 she will answer incorrectly. If the interaction is repeated k-times, Peggy will be caught with probability $1 - 2^{-k}$. The interaction between Peggy and Victor only reveals the fact that Peggy can detect a switch and not the color of the hat, making this zero-knowledge.

Verification Objectives: A zero-knowledge proof must satisfy three properties: *completeness* (an honest prover should be able to convince an honest verifier of a true statement), *soundness* (a cheating prover can convince an honest verifier with negligible probability) and *zero-knowledge* (no information apart from the veracity of the statement should be revealed). Completeness is a standard trace property, while zero-knowledge is the 2-safety property of indistinguishability. Consequently, the main challenge in automated verification of the zero-knowledge protocol described above is that of soundness. In this section, we discuss its specification and verification using quantitative hyperproperties.

$$\begin{array}{lll} X & \doteq & \{\mathsf{C}, \mathsf{P}, \mathsf{S}, i, \mathsf{R}\} \\ \mathit{Init}(X) & \doteq & (\forall i.\ 0 \leq \mathsf{C}[i] \leq 1) \wedge (\forall i.\ 0 \leq \mathsf{P}[i] \leq 1) \wedge \mathsf{S} \wedge (i = 1) \wedge (\mathsf{R} > 0) \\ \mathit{Tx}(X, X') & \doteq & (\mathsf{C}' = \mathsf{C}) \wedge (\mathsf{P}' = \mathsf{P}) \wedge (\mathsf{R}' = \mathsf{R}) \wedge \big(\mathsf{S}' = \big(\mathsf{S} \wedge (\mathsf{C}[i] = \mathsf{P}[i])\big)\big) \wedge \\ & & i' = \min(i + 1, \mathsf{R}) \end{array}$$

Fig. 1. Transition system model of the example protocol.

Soundness as a Quantitative Hyperproperty: Consider the transition system $M = \langle X, \mathit{Init}(X), \mathit{Tx}(X, X')\rangle$, shown in Fig. 1, representing this protocol. The variable R is a *parameter* of the system and refers to the number of rounds of the protocol. C and P are boolean arrays representing the challenges from the verifier to the prover, and the responses from the prover to the verifier, respectively. i is the current round, and S is a boolean flag that corresponds to whether the zero-knowledge proof has succeeded. C and P are initialized non-deterministically to model the fact that the verifier chooses their challenges randomly, and a cheating prover's best strategy is guessing. While a cheating prover can use any strategy, if the challenges are indistinguishable to her, then the best strategy is to sample responses from a uniform distribution.

Soundness is captured by the following quantitative hyperproperty (QHP):

$$\forall \pi_0 . \# \pi_1 {:} \mathbf{F}\,(\delta_{\pi_j, \pi_k}).\ \mathbf{G}\,(\psi_{\pi_0, \pi_1})\ \geq\ 2^{\mathsf{R}} - 1 \tag{1}$$

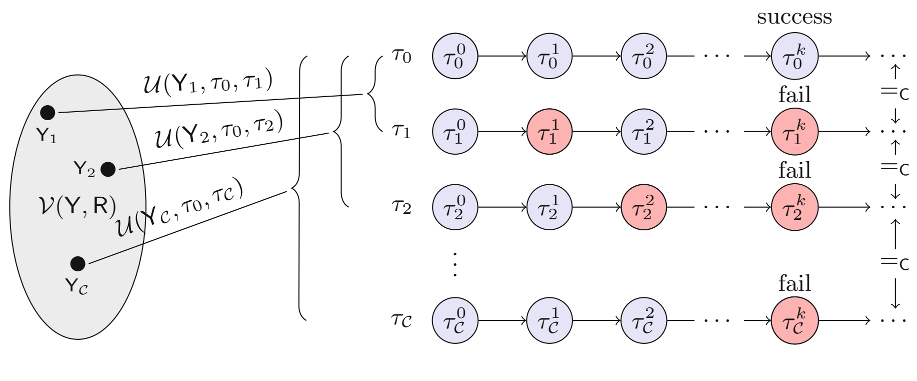
We will provide formal satisfaction semantics for QHPs in Sect. 3. For now, we informally describe its meaning. The term $\# \pi_1 {:} \mathbf{F}\,(\delta_{\pi_j, \pi_k}).\ \mathbf{G}\,(\psi_{\pi_0, \pi_1})\ \geq\ 2^{\mathsf{R}} - 1$ introduces a counting quantifier which stipulates the existence of at least $2^{\mathsf{R}} - 1$ traces satisfying certain conditions: (i) these traces must all be pairwise-different, where difference is defined by satisfaction of the formula $\mathbf{F}\,(\delta_{\pi_j, \pi_k})$ and (ii) all of these traces must be related to trace π_0 by the relation $\mathbf{G}\,(\psi_{\pi_0, \pi_1})$.

The state predicates δ and ψ are defined as follows.

$$\begin{array}{rcl} \delta(\sigma_1, \sigma_2) & \doteq & \sigma_1(\mathsf{P}[i]) \neq \sigma_2(\mathsf{P}[i]) \\ \psi(\sigma_1, \sigma_2) & \doteq & \big(\sigma_1((i = \mathsf{R}) \Rightarrow \mathsf{S}) \Rightarrow \sigma_2((i = \mathsf{R}) \Rightarrow \neg \mathsf{S})\big) \quad \wedge \\ & & \big(\sigma_1(\mathsf{C}) = \sigma_2(\mathsf{C}) \wedge \sigma_1(\mathsf{R}) = \sigma_2(\mathsf{R})\big) \end{array}$$

The requirement imposed by δ is that Peggy's responses be different at some step i for every pair of traces captured by the counting quantifier. ψ says that if trace π_0 is a trace where Peggy's cheating succeeds (i.e., $\mathsf{S} = \mathit{true}$ when $i = \mathsf{R}$), then in all traces captured by π_1, the challenges and number of rounds are the same as π_0 but Peggy's cheating is detected by Victor (i.e., $\mathsf{S} = \mathit{false}$ when $i = \mathsf{R}$). These requirements are illustrated in Fig. 2(b).

The QHP requires that for every trace in which a cheating prover succeeds in tricking the verifier for a given trace of challenges, there are $2^{\mathsf{R}} - 1$ other traces with the same challenges in which the prover's cheating is detected. Even though soundness is a probabilistic property over the distribution of the system's traces, it can be reduced to counting (and thus specified as a QHP) because each execution trace is sampled uniformly from a finite set. Therefore, if the QHP is satisfied, Peggy's probability of successful cheating is upper-bounded by $2^{-\mathsf{R}}$.


$\mathcal{U}(\mathsf{Y}_1, \tau_0, \tau_1)$
$\mathcal{U}(\mathsf{Y}_2, \tau_0, \tau_2)$
$\mathcal{U}(\mathsf{Y}_\mathcal{C}, \tau_0, \tau_\mathcal{C})$
Y_1
Y_2
$\mathcal{V}(\mathsf{Y}, \mathsf{R})$
$\mathsf{Y}_\mathcal{C}$
success
fail
fail
fail
$=_\mathsf{C}$

$$\begin{tikzcd}
\tau_0 & \tau_0^0 \arrow[r] & \tau_0^1 \arrow[r] & \tau_0^2 \arrow[r, dash] & \cdots \arrow[r] & \tau_0^k \arrow[r] & \cdots \\
\tau_1 & \tau_1^0 \arrow[r] & \tau_1^1 \arrow[r] & \tau_1^2 \arrow[r, dash] & \cdots \arrow[r] & \tau_1^k \arrow[r] & \cdots \\
\tau_2 & \tau_2^0 \arrow[r] & \tau_2^1 \arrow[r] & \tau_2^2 \arrow[r, dash] & \cdots \arrow[r] & \tau_2^k \arrow[r] & \cdots \\
& \vdots & & & & & \\
\tau_\mathcal{C} & \tau_\mathcal{C}^0 \arrow[r] & \tau_\mathcal{C}^1 \arrow[r] & \tau_\mathcal{C}^2 \arrow[r, dash] & \cdots \arrow[r] & \tau_\mathcal{C}^k \arrow[r] & \cdots
\end{tikzcd}$$

(a) Trace enumeration predicates. (b) Traces in the soundness QHP.

Fig. 2. Using trace enumeration predicates to verify the soundness QHP.

2.3 Solution Outline

To prove a QHP of the form $\forall \pi_0.\ \#\pi_1 : \Delta_{\pi_j,\pi_k}.\ \varphi\ \triangleleft\ N(Z)$, we construct a *trace enumeration predicate* $\mathcal{V}(\mathsf{Y}, Z)$ and show an injective/bijective mapping from assignments to Y in $\mathcal{V}(\mathsf{Y}, Z)$ and traces of the system. This allows us to prove $\forall \pi_0.\ \#\pi_1 \colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \triangleleft\ \#Y.\, \mathcal{V}(Y, Z)$. This part of the proof relies on the notion of a trace enumeration relation (Sect. 4). In the next step, we show that $\#Y.\, \mathcal{V}(Y, Z) \triangleleft N(Z)$ using the inference rules presented in Sect. 5. We now describe these steps in the context of the motivating example.

Verification of Soundness for the Z-K Hats Puzzle: Property 1 is illustrated in Fig. 2(b). τ_0 is a trace where the Z-K proof succeeds, while the proof fails for the set of traces $\Phi_\mathcal{C} = \{\tau_1, \tau_2, \ldots, \tau_\mathcal{C}\}$. The red states show the particular step of the proof in which an incorrect response is given by the prover, and each of these steps as well as their associated prover responses are pairwise different. The QHP is satisfied if $|\Phi_\mathcal{C}| \geq 2^{\mathtt{R}} - 1$ for every $\tau_0 \in \Phi_M$, where $\mathtt{R} = \tau_0^0(\mathsf{R})$.

The first step in our methodology is to construct a parameterized relation, called a trace enumeration relation, $\mathcal{U}(\mathsf{Y}, \tau_0, \tau_1)$. This relates τ_0 to each trace in the set $\Phi_\mathcal{C}$ and is parameterized by Y. For every value of the parameter Y, $\mathcal{U}$ relates a trace in which the proof succeeds (τ_0) to a trace in which the proof fails (τ_1). For every trace τ_0 in which the proof succeeds, the set $\{\tau_1 \mid \exists \mathsf{Y}.\, \mathcal{U}(\mathsf{Y}, \tau_0, \tau_1)\}$ corresponds to the set of traces with the same challenges and the same number of rounds, but with failed proofs of knowledge. Note this is a subset of $\Phi_\mathcal{C}$.

Next, we construct a predicate $\mathcal{V}(\mathsf{Y}, \mathsf{R})$ which defines valid assignments to $\mathcal{V}$ for a particular value of R. For a particular R, consider the set: $\{\sigma(\mathsf{Y}) \mid \sigma\ \models$

$\mathcal{V}(\mathsf{Y},\mathsf{R})\}$. Suppose we are able to show that the relation $\mathcal{U}$ is injective in Y and τ_0 for assignments to Y drawn from this set, then we can lower-bound the size of $\Phi_{\mathcal{C}}$ by the size of this set. In other words, we have reduced the problem of trace counting to the problem of counting assignments to $\mathcal{V}(\mathsf{Y},\mathsf{R})$.

Precisely stated, using $\mathcal{V}$ and $\mathcal{U}$, we show the following.

1. For every trace τ_0, and every assignment Y_i satisfying $\mathcal{V}(\mathsf{Y}_i, \tau_0^0(\mathsf{R}))$, there exists a corresponding trace τ_i that satisfies both $\mathcal{U}(\mathsf{Y}_i, \tau_0, \tau_i)$ and $\psi(\tau_0, \tau_i)$. (Note $\tau_0^0(\mathsf{R})$ refers to the valuation of R in the initial state of τ_0.)
2. Given two different satisfying assignments to $\mathcal{V}$ for a particular value of R, say Y_j and Y_k, the corresponding traces τ_j and τ_k are guaranteed to have different prover responses; in other words, the traces satisfy $\delta(\tau_j, \tau_k)$.

The above two properties, illustrated in Fig. 2(a), imply there is an injective mapping from satisfying assignments of $\mathcal{V}(\mathsf{Y},\mathsf{R})$ to traces in $\Phi_{\mathcal{C}}$. Therefore, the number of traces in $\Phi_{\mathcal{C}}$ can be lower bounded by the number of satisfying assignments to Y in $\mathcal{V}(\mathsf{Y},\mathsf{R})$, i.e. $\#\mathsf{Y}.\, \mathcal{V}(\mathsf{Y},\mathsf{R})$. We have reduced the difficult problem of counting traces into a slightly easier problem of counting satisfying assignments to a $FOL(\mathcal{T})$ formula.

The final step is to bound $\#\mathsf{Y}.\, \mathcal{V}(\mathsf{Y},\mathsf{R})$. For example, one well-known idea from enumerative combinatorics is that if a set A is the union of disjoint sets B and C, then $|A| = |B| + |C|$. Translated to model counting, the above can be written as $\#X.\, F(X,Y) = \#X.\, G(X,Y) + \#X.\, H(X,Y)$ if $F(X,Y) \Leftrightarrow G(X,Y) \vee H(X,Y)$ is valid and $G(X,Y) \wedge H(X,Y)$ is unsat.[2] We present a set of inference rules in Sect. 5 that build on this and related ideas. These inference rules allow us derive a machine-checked proof of the bound $\#\mathsf{Y}.\, \mathcal{V}(\mathsf{Y},\mathsf{R}) \geq 2^{\mathsf{R}} - 1$, thus completing the proof of Property 1 for the Z-K hats puzzle.

3 Overview of Quantitative Hyperproperties

This section introduces a logic for the specification of quantitative hyperproperties over symbolic transition systems. We present satisfaction semantics for this logic and then discuss its applications in security verification.

$$\begin{array}{lll}
\psi & ::= & \forall \pi.\ \psi \mid \#\pi{:}\, \Delta_{\pi_j,\pi_k}.\ \psi \ \lhd\ N(Z) \mid \varphi \\
\varphi & ::= & \mathcal{P}_{\pi_1,\pi_2,\ldots,\pi_k} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \mathbf{U}\, \varphi \mid \mathbf{X}\, \varphi \\
\lhd & ::= & \leq \mid = \mid \geq
\end{array}$$

Fig. 3. Grammar of Quantitative HyperLTL.

[2] We note there is an implied universal quantifier here. To be precise, we must write $\forall Y.\ \#X.\, F(X,Y) = \#X.\, G(X,Y) + \#X.\, H(X,Y)$.

3.1 Quantitative Hyperproperties

Figure 3 shows the syntax of Quantitative HyperLTL, our extension of HyperLTL [30] that allows specification of quantitative hyperproperties over symbolic transition systems. There are two noteworthy differences from the presentation of HyperLTL in [30]. The first is the predicate $\mathcal{P}_{\pi_1,\pi_2,\ldots,\pi_k}$. This refers to a k-ary state predicate $\mathcal{P}$ that is applied to the first element of each trace in the subscript. These are analogous to atomic propositions in presentations that use Kripke structures and are defined as k-ary state predicates to capture relational properties over traces of the transition system. For example, consider the predicate $\mathcal{P}(\sigma_0, \sigma_1) \doteq (\mathit{input}(\sigma_0) = \mathit{input}(\sigma_1))$. Given this definition, a system M with exactly two traces $\varPhi_M = \{\tau_1, \tau_2\}$ satisfies the HyperLTL formula $\forall \pi_1, \pi_2.\ \mathcal{P}_{\pi_1,\pi_2}$ iff $\mathit{input}(\tau_1^0) = \mathit{input}(\tau_2^0)$. This hyperproperty requires that the input in the initial state of the system be deterministically initialized.

The second difference is the new *counting quantifier*: $\#\pi \colon \varDelta_{\pi_j,\pi_k}.\ \psi \lhd N(Z)$.[3] $\varDelta_{\pi_j,\pi_k}$ is an unquantified HyperLTL formula over two "fresh" trace variables π_j and π_k that encodes when two traces are considered different. ψ is another (possibly-quantified) HyperLTL formula. The operator $\lhd$ can be $\leq$, $=$, or $\geq$. $N(Z)$ is an integer-sorted term in $\mathit{FOL}(\mathcal{T})$ over the variables in the set Z, $Z \subset X$ where X is the set of state variables of the transition system under consideration. Z typically refers to the subset of the state variables that define the parameters of the transition system; e.g. $Z = \{\mathsf{R}\}$ for the Z-K proof transition system in Fig. 1, the number of blocks in a model of Path ORAM, the size of an array, etc. Typically, the variables in the set Z do not change after initialization. Informally stated, the counting quantifier is satisfied if a maximally large set $\varPhi_{\mathcal{C}} \subseteq \varPhi$, satisfying the two conditions below, has cardinality $\lhd$ *count* where *count* is the valuation of $N(Z)$ in the initial state of every trace in $\varPhi_{\mathcal{C}}$. Those conditions are: (i) each of the traces in $\varPhi_{\mathcal{C}}$ are pairwise different as defined by satisfaction of $\varDelta_{\pi_j,\pi_k}$, and (ii) every trace in this set satisfies the HyperLTL formula ψ.

The remaining operators are standard, so we do not discuss them further and instead provide formal satisfaction semantics.

Satisfaction Semantics of Quantitative HyperLTL The validity judgement of a property φ by a set of traces $\varPhi$ is defined with respect to a trace assignment $\varPi : \mathit{Vars} \rightarrow \varPhi$. Here, Vars is the set of trace variables. We use $\pi, \pi_1, \pi_2, \ldots$ to refer to trace variables.[4] The partial function $\varPi$ is a mapping from trace variables to traces. We use the notation $\varPi[\pi \mapsto \tau]$ to refer to a trace assignment that is identical to $\varPi$ except for the trace variable π which now maps to the trace τ. We write $\varPi \models_{\varPhi} \psi$ if the set of traces $\varPhi$ satisfies the property ψ under the trace assignment $\varPi$. We will drop the subscript $\varPhi$ from $\models_{\varPhi}$ if it is clear from the context or irrelevant. The notation $\varPi^{[i,\infty]}$ is an abbreviation

[3] A counting quantifier over Kripke structures was introduced by Finkbeiner et al. [29]. Our definition is slightly different and a detailed comparison is deferred to Sect. 7.

[4] Note the distinction between trace variables denoted by π_1, π_2, etc. and traces which are denoted by τ_1, τ_2, etc.

for the new trace assignment obtained by taking the suffix starting from index i of every trace in Π: $\Pi^{[i,\infty]}(\pi) = \Pi(\pi)^{[i,\infty]}$ for every trace $\pi \in dom(\Pi)$ where $dom(\Pi)$ is the domain of Π. We write $\Pi \not\models_{\Phi} \psi$ when $\Pi \models_{\Phi} \psi$ is not satisfied. Satisfaction rules for HyperLTL formulas are shown in Fig. 4.

$$
\begin{array}{ll}
\Pi \models_{\Phi} \forall\pi.\ \psi & \text{iff for all } \tau \in \Phi : \Pi[\pi \mapsto \tau] \models_{\Phi} \psi \\
\Pi \models_{\Phi} \#\pi{:}\, \Delta_{\pi_j,\pi_k}.\ \psi \ \triangleleft\ N(Z) & \text{iff } |\Phi_{\mathcal{C}}| = 0 \Rightarrow 0 \triangleleft N(Z) \text{ is valid, and} \\
& \quad |\Phi_{\mathcal{C}}| > 0 \Rightarrow \forall\tau \in \Phi_{\mathcal{C}}.\ |\Phi_{\mathcal{C}}| \triangleleft \tau^0(N(Z)), \text{ where,} \\
& \quad \Phi_{\mathcal{C}} \subseteq \Phi \text{ is a maximally large set such that:} \\
& \quad \forall\tau_j, \tau_k \in \Phi_{\mathcal{C}}. \\
& \qquad\qquad \tau_j \neq \tau_k \Leftrightarrow \{\pi_j \mapsto \tau_j, \pi_k \mapsto \tau_k\} \models \Delta_{\pi_j,\pi_k} \\
& \quad \text{and, } \forall\tau \in \Phi_{\mathcal{C}}.\ \Pi[\pi \mapsto \tau] \models_{\Phi} \psi \\
\Pi \models_{\Phi} \mathcal{P}_{\pi_1,\ldots,\pi_k} & \text{iff } \mathcal{P}(\Pi(\pi_1)^0, \ldots, \Pi(\pi_k)^0) \text{ is valid} \\
\Pi \models_{\Phi} \neg\psi & \text{iff } \Pi \not\models_{\Phi} \psi \\
\Pi \models_{\Phi} \psi \vee \varphi & \text{iff } \Pi \models_{\Phi} \psi \text{ or } \Pi \models_{\Phi} \varphi \\
\Pi \models_{\Phi} \mathbf{X}\,\varphi & \text{iff } \Pi^{[1,\infty]} \models_{\Phi} \varphi \\
\Pi \models_{\Phi} \varphi\,\mathbf{U}\,\psi & \text{iff there exists } j \geq 0 : \Pi^{[j,\infty]} \models_{\Phi} \psi \\
& \quad \text{and for all } 0 \leq i < j : \Pi^{[i,\infty]} \models_{\Phi} \varphi
\end{array}
$$

Fig. 4. Satisfaction semantics for Quantitative HyperLTL formulas over symbolic transition systems.

Definition 2 (Quantitative HyperLTL Satisfaction). *We say that the transition system M satisfies the property ψ, denoted by $M \models \psi$ if the empty trace assignment $\emptyset$ satisfies formula ψ for the set of traces Φ_M, that is $\emptyset \models_{\Phi_M} \psi$.*

Additional Operators: The above showed the minimal set of operators required in Quantitative HyperLTL. The rest of this paper will use the other standard operators such as $\wedge$ (conjunction), $\Rightarrow$ (implication), $\mathbf{F}$ (future/eventually) and $\mathbf{G}$ (globally/always) which can be defined in terms of the operators in Fig. 3.

Well-Defined Formulas: In order for the semantics of Quantified HyperLTL to be meaningful, we need certain semantic restrictions on the structure of QHPs.

Definition 3 (Well-defined QHPs). *An instance of a counting quantifier $\#\pi{:}\, \Delta_{\pi_j,\pi_k}.\ \varphi \ \triangleleft\ N(Z)$ is said to be well-defined if:*

1. *$\neg\Delta_{\pi_j,\pi_k}$ is an equivalence relation over the set of all traces Φ, and*
2. *In every set of the traces $\Phi_{\mathcal{C}}$ captured by the counting quantifier in the semantics shown in Fig. 4, the term $N(Z)$ has the same valuation for all initial states: $\forall\tau_i, \tau_j \in \Phi_{\mathcal{C}}.\ \tau_i^0(N(Z)) = \tau_j^0(N(Z))$.*

A Quantified HyperLTL formula is said to be well-defined if every instance of a counting quantifier in the formula is well-defined.

Example 1 (Well-defined QHPs). The QHPs presented in the rest of this paper are all well-defined, so here we give an example of a QHP that is *not* well-defined. Consider this variant of Property 1: $\forall\pi_0.\#\pi_1\colon true.\ \mathbf{G}\,(\psi_{\pi_0,\pi_1}) \ \geq\ 2^{\mathsf{R}} - 1$. This is not a well-defined QHP because Δ_{π_j,π_k} in the counting quantifier is simply *true*, and its negation is not an equivalence relation over the set of traces.

Note that condition (1) in the definition above affects Δ_{π_j,π_k} while condition (2) places a restriction on φ. The former condition prevents double-counting of traces, while the latter ensures that the trace count is unambiguous.

The properties in our experiments require only syntactic checks to verify well-definedness. Specifically, Δ_{π_j,π_k} is always of the form $\mathbf{F}\,(\mathcal{P}_{\pi_j,\pi_k})$ where $\mathcal{P}$ is of the form $\mathcal{P}(\sigma_1,\sigma_2) \doteq f(\sigma_1) \neq f(\sigma_2)$. The negation of this is obviously an equivalence relation over the set of all traces. Secondly, our QHPs are of the form $\forall\pi_0.\ \#\pi_1\colon\Delta_{\pi_j,\pi_k}.\ \varphi\ \triangleleft\ N(Z)$ where φ enforces equality of the variables in Z between the traces π_0 and π_1. These two features guarantee well-definedness. In the rest of this paper, we only consider well-defined QHPs.

3.2 Applications of QHPs in Security Specification

Deniability: Our first example of a quantitative hyperproperty is deniability. Suppose $obs(\sigma)$ is a term that corresponds to the adversary observable part of the state σ, while $secret(\sigma)$ corresponds to the secret component of the state σ. Deniability is satisfied when every trace of adversary observations can be generated by at least $N(Z)$ different secrets. For this, we define $\delta(\sigma_1,\sigma_2) \doteq secret(\sigma_1) \neq secret(\sigma_2)$ and $\approx^O(\sigma_1,\sigma_2) \doteq obs(\sigma_1) = obs(\sigma_2)$.

$$\forall\pi_0.\#\pi_1\colon\mathbf{F}\,(\delta_{\pi_j,\pi_k}).\ \mathbf{G}\,(\approx^O_{\pi_0,\pi_1})\ \geq\ N(Z)$$

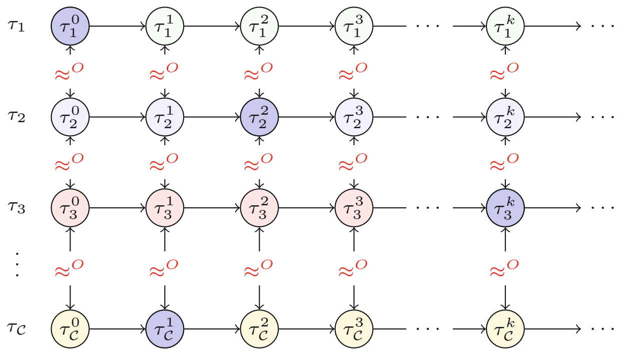

τ_1
τ_2
τ_3
$\tau_{\mathcal{C}}$
τ_1^0 τ_1^1 τ_1^2 τ_1^3 τ_1^k
τ_2^0 τ_2^1 τ_2^2 τ_2^3 τ_2^k
τ_3^0 τ_3^1 τ_3^2 τ_3^3 τ_3^k
$\tau_{\mathcal{C}}^0$ $\tau_{\mathcal{C}}^1$ $\tau_{\mathcal{C}}^2$ $\tau_{\mathcal{C}}^3$ $\tau_{\mathcal{C}}^k$
$\approx^O$
...

Fig. 5. Illustrating deniability.

Figure 5 illustrates deniability. It shows a set of traces $\Phi_{\mathcal{C}} := \{\tau_1, \tau_2, \ldots, \tau_{\mathcal{C}}\}$; the circles represent the states in each trace and the secret values are shown by color of the circle. For these traces, every pair of corresponding states have the same observations: represented by $\approx^O$, and every distinct pair of traces differ in the secrets. Deniability is satisfied if $|\Phi_{\mathcal{C}}| \geq N(Z)$. Satisfaction implies that every trace of adversary observations has at least $N(Z)$ counterparts with identical observations but different values of $\mathit{secret}(\sigma)$. If we can show in a system satisfying deniability that each trace of secrets is equiprobable and $N(Z)$ grows exponentially in some parameters of the system, then we can conclude that the system satisfies computational indistinguishability. Deniability can capture probabilistic notions of confidentiality, such as confidentiality of Path ORAM.

Soundness: While deniability encodes a form of confidentiality, soundness is its dual in the context of integrity. One example of soundness was given in Sect. 2.2 for the Z-K hats puzzle. Soundness is generally applicable to protocols that offer probabilistic integrity guarantees. For instance, many interactive challenge-response protocols consist of repeated rounds such that if the prover succeeds in all rounds, the verifier can be convinced with a high probability that the prover is not cheating. This can be viewed as a QHP stating that for every trace in which a dishonest prover tricks a verifier into accepting an invalid proof, there are at least $N(Z)$ other traces with different prover responses in which the cheating is detected. As usual, we require that traces be uniformly sampled from a finite set in order to state soundness as a QHP.

Soundness is stated as $\forall \pi_0 . \# \pi_1 \colon \mathbf{F}\,(\delta_{\pi_j,\pi_k}).\ \mathbf{G}\,(\psi_{\pi_0,\pi_1})\ \geq\ N(Z)$. The relation δ is defined as two states having different prover responses. ψ requires the challenge-response protocol to fail in π_1 if it succeeded in π_0 and also that the system parameters (the variables in Z) be identical between π_0 and π_1.

Summarizing QHP Specification: These examples demonstrate that QHPs have important applications in security verification. They capture probabilistic notions of both confidentiality and integrity. In particular, the following form of QHPs consisting of a single quantifier alternation seems especially relevant for security verification: $\forall \pi_0.\ \# \pi_1 \colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \triangleleft\ N(Z)$. Each of the examples of quantitative hyperproperties discussed in the previous subsection – deniability, soundness, as well as others like quantitative non-interference [46,54] fit in this template. Therefore, in the rest of this paper, we focus on developing scalable verification techniques for QHPs that follow this template.

4 Trace Enumerations

This section introduces the notion of a trace enumeration, which is a technique that allows us to reduce the problem of counting traces to that of counting satisfiable assignments to a formula in $\mathit{FOL}(\mathcal{T})$.

4.1 Trace Enumeration Relations

We now formalize injective trace enumerations, which allow us to lower-bound the number of traces captured by a counting quantifier in a QHP.

Definition 4 (Injective Trace Enumeration). *Let us consider a transition system $M = \langle X, Init(X), Tx(X, X')\rangle$ and the relation $\mathcal{U}(Y, \tau_1, \tau_2)$ where Y is a set of variables disjoint from X, τ_1 and τ_2 are traces of this transition system. Let $\forall\pi_0.\ \#\pi_1\colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \geq\ N(Z)$ be a QHP where $Z \subset X$. Suppose $\mathcal{V}(Y, Z)$ is a predicate over the variables in Y and Z. We say that the pair $\mathcal{V}(Y, Z)$ and $\mathcal{U}(Y, \tau_1, \tau_2)$ form an injective trace enumeration of the system M for the QHP $\forall\pi_0.\ \#\pi_1\colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \geq\ N(Z)$ iff the following conditions are satisfied:*

1. *For every trace τ_0 in Φ_M and every satisfying assignment $(\mathtt{Y}, \mathtt{Z})$ for the predicate $\mathcal{V}(Y, Z)$, there exists a trace $\tau_1 \in \Phi_M$ which is related to the trace τ_0 as per the relation $\mathcal{U}$ via this same assignment to Y. Further, the pair τ_0 and τ_1 satisfy the property φ and the valuation of the variables in Z in the initial state of τ_1 is equal to $\mathtt{Z}$.*

$$\begin{aligned}&\forall\tau_0 \in \Phi_M, \mathtt{Y}, \mathtt{Z}.\ \mathcal{V}(\mathtt{Y}, \mathtt{Z}) \Rightarrow \\ &\quad\big(\exists\tau_1 \in \Phi_M.\ \mathcal{U}(\mathtt{Y}, \tau_0, \tau_1) \wedge \{\pi_0 \mapsto \tau_0, \pi_1 \mapsto \tau_1\} \models \varphi \wedge \tau_1^0(Z) = \mathtt{Z}\big)\end{aligned} \tag{2}$$

2. *Different assignments to the variables in Y for the formula $\mathcal{V}(Y, Z)$ enumerate different traces in $\mathcal{U}(Y, \tau_0, \tau_1)$, where "different" means satisfaction of Δ_{π_j,π_k}.*

$$\begin{aligned}&\forall\tau_0, \tau_1, \tau_2 \in \Phi_M, \mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Z}. \\ &\quad\mathcal{V}(\mathtt{Y}_1, \mathtt{Z}) \wedge \mathcal{V}(\mathtt{Y}_2, \mathtt{Z}) \wedge \mathtt{Y}_1 \neq \mathtt{Y}_2 \qquad \Rightarrow \\ &\quad\mathcal{U}(\mathtt{Y}_1, \tau_0, \tau_1) \wedge \mathcal{U}(\mathtt{Y}_2, \tau_0, \tau_2) \wedge \tau_1^0(Z) = \mathtt{Z} \wedge \tau_2^0(Z) = \mathtt{Z} \qquad \Rightarrow \\ &\quad\{\pi_j \mapsto \tau_1, \pi_k \mapsto \tau_2\} \models \Delta_{\pi_j,\pi_k}\end{aligned} \tag{3}$$

If $\mathcal{V}$ and $\mathcal{U}$ form an injective trace enumeration M for the property $\forall\pi_0.\ \#\pi_1\colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \geq\ N(Z)$, then for every trace τ_0, there exist at least as many traces satisfying the counting quantifier as there are satisfying assignments to Y in $\mathcal{V}(Y, Z)$. This is made precise in the following lemma.

Lemma 1. *[Trace Count Lower-Bound] If $\mathcal{V}(Y, Z)$ and $\mathcal{U}(Y, \tau_1, \tau_2)$ form an injective trace enumeration of the system M for the QHP $\forall\pi_0.\ \#\pi_1\colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \geq\ N(Z)$ and if $\#Y.\ \mathcal{V}(Y, Z)$ is finite for all assignments to Z, then $M \models \forall\pi_0.\#\pi_1\colon \Delta_{\pi_j,\pi_k}.\ \varphi\ \geq\ \#Y.\ \mathcal{V}(Y, Z)$.*

Example 2 (Injective Trace Enumeration). Let $\mathsf{P}_0[1], \ldots, \mathsf{P}_0[\mathsf{R}]$ be a trace of correct responses for some particular sequence of challenges for our running example. Consider the array $\mathsf{Y}[1], \mathsf{Y}[2], \ldots, \mathsf{Y}[\mathsf{R}]$ where each $\mathsf{Y}[j] \in \{0, 1\}$. Y is a boolean array of size R, and $\mathsf{Y}[i] = 1$ means that the prover gives an incorrect response to the challenge in round i. We can define the predicate $\mathcal{V}$ as follows.

$$\mathcal{V}(\mathsf{Y}, \mathsf{R}) \doteq \big(\exists i.\ 1 \leq i \leq \mathsf{R} \wedge \mathsf{Y}[i] \neq 0\big) \wedge \big(\forall i.\ (i < 1 \vee i > \mathsf{R}) \Rightarrow \mathsf{Y}[i] = 0\big) \tag{4}$$

The above definition ensures that at least one response is incorrect. Notice that for every assignment to Y except the assignment of all zeros, the trace of responses defined by $\forall j.\ \mathsf{P}_1[j] = \mathsf{P}_0[j] \oplus \mathsf{Y}[j]$ (where $\oplus$ is exclusive or) corresponds to a valid trace of the system and satisfies the counting quantifier in Property 1. Specifically, every such response from the prover is incorrect and will result in the protocol failing. We can use the above facts to define the relation $\mathcal{U}$ as follows:

$$\begin{aligned}\mathcal{U}(\mathsf{Y}, \tau_1, \tau_2) \doteq \big(\forall j.\ \tau_1^0(\mathsf{P}[j]) = \tau_2^0(\mathsf{P}[j]) \oplus \mathsf{Y}[j]\big) \quad & \wedge \qquad (5)\\ \tau_1^0(\mathsf{C}) = \tau_2^0(\mathsf{C}) \wedge \tau_1^0(\mathsf{R}) = \tau_2^0(\mathsf{R}) \wedge (\tau_1^{\mathsf{R}}(S) \Rightarrow \neg\tau_2^{\mathsf{R}}(S))\end{aligned}$$

The pair $\mathcal{V}$ and $\mathcal{U}$ form an injective trace enumeration for the system M (defined in Fig. 1) for the Property 1. This is because different Y's will result in different prover responses for the same challenges. By Lemma 1, we can conclude that Property 1 is satisfied if $\#\mathsf{Y}.\, \mathcal{V}(\mathsf{Y}, \mathsf{R}) \geq 2^{\mathsf{R}} - 1$

Analogous to injective trace enumerations, it is also possible to define surjective trace enumerations that upper-bound the number of traces captured by a counting quantifier. Details of surjective trace enumerations are presented in the extended version of the paper [43].

5 Model Counting

As discussed in the previous section, trace enumeration relations can bound the number of satisfying traces in a QHP. Given a QHP $\forall\pi_0.\ \#\pi_1 : \Delta_{\pi_j,\pi_k}.\ \varphi \ \triangleleft\ N(Z)$, appropriate trace enumeration predicates $\mathcal{V}(Y, Z)$ and $\mathcal{U}$ can be used to derive that $\forall\pi_0.\ \#\pi_1{:}\,\Delta_{\pi_j,\pi_k}.\ \varphi \ \triangleleft\ \#Y.\,\mathcal{V}(Y, Z)$. The final step in our verification methodology is to show validity of $\#Y.\,\mathcal{V}(Y, Z) \triangleleft N(Z)$. To that end, this section discusses our novel technique for model counting.

5.1 Model Counting via SMT Solving

Our approach borrows ideas from enumerative combinatorics [13,52,56] and introduces the inference rules shown in Fig. 6 to reason about model counts for formulas in $FOL(\mathcal{T})$. Each of the conclusions in the inference rules is a statement involving model counts of $FOL(\mathcal{T})$ formulas, while each of the premises is a formula in $FOL(\mathcal{T})$ that *does not involve model counts* and can, therefore, be checked using SAT/SMT solvers. Most of the rules are straightforward, and we do not describe them due to space constraints. The three interesting rules – *Injectivity*, $Ind_{\leq}$ and $Ind_{\geq}$ – are discussed below.

Injectivity: This rule is based on the following idea from enumerative combinatorics. Suppose we have two sets A and B. We can show that $|A| \leq |B|$ if there exists an injective function from A to B. Translating this to model counts, the set A in the rule corresponds to satisfying assignments to $f(X)$, B corresponds to satisfying assignments to $g(Y)$ and $\mathscr{F}$ is the injective witness function.

$Ind_{\geq}$ and $Ind_{\leq}$: Suppose the formulas $f(X,n)$ and $g(Y,n)$ are parameterized by the integer variable n. If an injective witness function $\mathscr{G}(X,Y,n)$ is able to "lift" satisfying assignments of $f(X_n,n)$ and $g(Y_n,n)$ into a satisfying assignment of $f(X_{n+1},n+1)$, then we can conclude that the number of satisfying assignments to $f(X,n+1)$ are at least as many as the product of the number of satisfying assignments to $f(X,n)$ and $g(Y,n)$. $Ind_{\leq}$ is the surjective version of this rule. It applies when a satisfying assignment to $f(X_{n+1},n+1)$ can be "lowered" into satisfying assignments to $f(X_n,n)$ and $g(Y_n,n)$ where the values of X_n and Y_n are given by the witness functions $\mathscr{H}_x$ and $\mathscr{H}_y$ respectively.

$$\frac{}{(\#i.\, a \leq i < b) \;=\; \max(b-a, 0)}\;\textit{Range} \qquad \frac{}{\#Y.\, f(X) \;\geq\; 0}\;\textit{Positive}$$

$$\frac{\bigwedge_{i=1}^{c} f(X_i) \wedge \textit{distinct}(X_1,\ldots,X_c) \text{ is } \mathsf{sat}}{\#X.\, f(X) \;\geq\; c}\;\textit{ConstLB}$$

$$\frac{\bigwedge_{i=1}^{c} f(X_i) \wedge \textit{distinct}(X_1,\ldots,X_c) \text{ is } \mathsf{unsat}}{\#X.\, f(X) \;<\; c}\;\textit{ConstUB}$$

$$\frac{f(X,Y) \Rightarrow g(X,Y)}{\#X.\, f(X,Y) \;\leq\; \#X.\, g(X,Y)}\;\textit{UB}$$

$$\frac{h(X,Y) \Leftrightarrow f(X) \wedge g(Y)}{\#X \cup Y.\, h(X,Y) \;\leq\; \#X.\, f(X) \times \#Y.\, g(Y)}\;\textit{AndUB}$$

$$\frac{\begin{array}{c} f(X) \Rightarrow g(\mathscr{F}(X)) \\ \big(f(X_1) \wedge f(X_2) \wedge X_1 \neq X_2\big) \Rightarrow \mathscr{F}(X_1) \neq \mathscr{F}(X_2) \end{array}}{\#X.\, f(X) \;\leq\; \#Y.\, g(Y)}\;\textit{Injectivity}$$

$$\frac{h(X,Y) \Leftrightarrow f(X) \wedge g(Y) \quad X \cap Y = \emptyset}{\#X \cup Y.\, h(X,Y) \;=\; \#X.\, f(X) \times \#Y.\, g(Y)}\;\textit{Disjoint}$$

$$\frac{f(X,Y) \Leftrightarrow g(X,Y) \vee h(X,Y)}{\#X.\, f(X,Y) \;=\; \#X.\, g(X,Y) + \#X.\, h(X,Y) - \#X.\, \big(g(X,Y) \wedge h(X,Y)\big)}\;\textit{Or}$$

$$\frac{\begin{array}{c} \big(f(X,n) \wedge g(Y,n)\big) \Rightarrow f(\mathscr{G}(X,Y,n), n+1) \\ (X_1 \neq X_2 \vee Y_1 \neq Y_2) \Rightarrow \mathscr{G}(X_1,Y_1,n) \neq \mathscr{G}(X_2,Y_2,n) \end{array}}{\#X.\, f(X,n+1) \;\geq\; \#X.\, f(X,n) \times \#Y.\, g(Y,n)}\;\textit{Ind}_{\geq}$$

$$\frac{\begin{array}{c} f(X,n+1) \Rightarrow \big(f(\mathscr{H}_x(X,n+1),n) \wedge g(\mathscr{H}_y(X,n+1),n)\big) \\ X_1 \neq X_2 \Rightarrow \big(\mathscr{H}_x(X_1,n) \neq \mathscr{H}_x(X_2,n) \vee \mathscr{H}_y(Y_1,n) \neq \mathscr{H}_y(Y_2,n)\big) \end{array}}{\#X.\, f(X,n+1) \;\leq\; \#X.\, f(X,n) \times \#Y.\, g(Y,n)}\;\textit{Ind}_{\leq}$$

Fig. 6. Model counting proof rules. Unless otherwise specified, premises are satisfied when the formula is valid. Conclusions have an implicit universal quantifier.

5.2 Model Counting in the Motivating Example

The definition of the predicate $\mathcal{V}$ in the motivating example is shown below.

$$\mathcal{V}(\mathsf{Y},\mathsf{R}) \doteq \big(\exists i.\ 1 \le i \le \mathsf{R} \wedge \mathsf{Y}[i] \neq 0\big) \wedge \big(\forall i.\ ((i < 1 \vee i > \mathsf{R}) \Rightarrow \mathsf{Y}[i] = 0)\big)$$

Our task is to show $\#\mathsf{Y}.\,\mathcal{V}(\mathsf{Y},\mathsf{R}) = 2^{\mathsf{R}} - 1$. Recall that Y is an array of binary values (i.e. the integers 0 and 1) and consider the following predicates: $\mathcal{V}_f(\mathsf{Y},\mathsf{R}) \doteq \big(\forall i.\ (i < 1 \vee i > \mathsf{R}) \Rightarrow \mathsf{Y}[i] = 0\big)$, $\mathcal{V}_1(\mathsf{Y},\mathsf{R}) \doteq \big(\forall i.\ \mathsf{Y}[i] = 0\big)$ and $\mathcal{W}(i) \doteq 0 \le i < 2$. Using these definitions, the proof is as follows.

1. (*ConstUB*, *Positive*) $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) \wedge \mathcal{V}_1(\mathsf{Y},\mathsf{R}) = 1$.
2. (*Or*) $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) = \#\mathsf{Y}.\,\mathcal{V}(\mathsf{Y},\mathsf{R}) + \#\mathsf{Y}.\,\mathcal{V}_1(\mathsf{Y},\mathsf{R})$.
3. (*ConstLB*, *ConstUB*) $\#\mathsf{Y}.\,\mathcal{V}_1(\mathsf{Y},\mathsf{R}) = 1$.
4. (*ConstLB*, *ConstUB*) $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},1) = 2$.
5. ($Ind_{\le}$): $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) \le \#i.\,\mathcal{W}(i) \times \#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}-1)$.
6. ($Ind_{\ge}$): $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) \ge \#i.\,\mathcal{W}(i) \times \#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}-1)$.
7. (*Range*): $\#i.\,\mathcal{W}(i) = 2$.
8. $(4-7)$ imply that $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) = 2 \times \#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}-1)$, $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},1) = 2$, this means $\#\mathsf{Y}.\,\mathcal{V}_f(\mathsf{Y},\mathsf{R}) = 2^{\mathsf{R}}$.
9. (2, 3, 8) imply that $\#\mathsf{Y}.\,\mathcal{V}(\mathsf{Y},\mathsf{R}) = 2^{\mathsf{R}} - 1$.

In step 5, the witness function is $\mathscr{G}(\mathsf{Y},\mathsf{R},i) \doteq \mathsf{Y}[\mathsf{R}+1 \mapsto i]$, while in step 6, they are $\mathscr{H}_{\langle \mathsf{Y},\mathsf{R}\rangle}(\mathsf{Y},\mathsf{R}+1) \doteq \langle \mathsf{Y}[\mathsf{R}+1 \mapsto 0], \mathsf{R}\rangle$ and $\mathscr{H}_i(\mathsf{Y},\mathsf{R}+1) \doteq (\mathsf{Y}[\mathsf{R}+1])$.[5] Note steps 8 and 9 are automatically discharged by the SMT solver.

6 Experimental Results and Discussion

In this section, we present an experimental evaluation of the use of trace enumerations for the verification of quantitative hyperproperties.

6.1 Methodology

We studied five systems with varying complexity and QHPs. These were modeled in the UCLID5 modeling and verification framework [44,51], which uses the Z3 SMT solver (v4.8.6) [23] to discharge the proof obligations. The experiments were run on an Intel i7-4770 CPU @ 3.40 GHz with 8 cores and 32 GB RAM.

The verification conditions are currently manually generated from the models, but automation of this is straightforward and ongoing. The k-trace properties were proven using self-composition [9,10] and induction. A number of strengthening invariants had to be specified manually for the inductive proofs. Many of the invariants are relational *and* quantified and, therefore, difficult to infer algorithmically. We note that recent work has made progress toward automated inference of quantified invariants [27,36].

[5] The notation $arr[i \mapsto v]$ denotes an array that is identical to arr except for index i which contains v.

6.2 Overview of Results

Due to limited space, we only provide a brief description of our benchmarks for evaluation and refer the interested reader to the extended version of our paper [43] for a more detailed discussion. We have also made the models and associated proof scripts available at [25]. A brief overview of the case studies follows.

Table 1. Verification results of models.

Benchmark	Hyperproperty	Model LoC	Proof LoC	Num. Annot	Verif. Time
Electronic purse [7]	Deniability	46	93	9	3.92 s
Password checker [29]	Quantitative non-interference	59	100	10	4.69 s
F-Y array shuffle	Quantitative information flow	86	195	96	7.38 s
ZK hats (Sect. 2.2)	Soundness	91	191	36	6.34 s
Path ORAM [48]	Deniability	587	209	142	9.74 s

1. **Electronic Purse.** We model an electronic purse, with a secret initial balance, proposed by Backes et al. [7]. A fixed amount is debited from the purse until the balance is insufficient for the next transaction. We prove a deniability property: there is a sufficient number of traces with identical attacker observations but different initial balances.
2. **Password Checker.** We model the password checker from Finkbeiner et al. [29], but we allow passwords of unbounded length n. We prove quantitative non-interference: information leakage to an attacker is $\leq n$ bits.
3. **Array Shuffle.** We implement a variant of the Fisher-Yates shuffle. We chose this because producing random permutations of an array is an important component of certain cryptographic protocols (e.g., Ring ORAM [40]). We prove a quantitative information flow property stating that all possible permutations are indeed generated by the shuffling algorithm.
4. **ZK Hats.** We prove soundness of the zero-knowledge protocol in Sect. 2.
5. **Path ORAM.** Discussed in Sect. 6.3.

The properties we prove on these models and the results of our evaluation are presented in Table 1 which shows the size of each model, the number of lines of proof code (this is the code for self-composition, property specification, etc.), the number of verification annotations (invariants and procedure pre-/post-conditions) and the verification time for each example. Once the auxiliary strengthening invariants are specified, the verification completes within a few seconds. This suggests that the methodology can scale to larger models, and even implementations. The main challenge in the application of the methodology is the construction of the trace enumeration relations, associated witness functions, and the specification of strengthening invariants. Each of these requires application-specific insight. Since most of our enumerations and invariants are quantified, some of the proofs also required tweaking the SMT solver's configuration options (e.g. turning off model-based quantifier instantiation in Z3).

6.3 Deniability of Path ORAM

In this section, we discuss our main case study: the application of trace enumerations for verifying deniability of server access patterns in Path ORAM [48], a practical variant of Oblivious RAM (ORAM) [33]. ORAMs refer to a class of algorithms that allow a client with a small amount of storage to store/load a large amount of data on an untrusted server while concealing the client access pattern from the server. Path ORAM stores encrypted data on the server in an augmented binary tree format. Each node stores Z data blocks, referred to as *buckets* of size Z. Additionally, the client has a small amount of local storage called the *stash*. The client maintains a secret mapping called the *position map* to keep track of the path where a data block is stored on the server. Each entry in the position map maps a client address to a leaf on the server. Path ORAM maintains the invariant that every block is stored somewhere along the path from the root to the leaf node that the block is mapped to by the position map.

Deniability of Server Access Patterns in Path ORAM: We formulate security of access patterns in Path ORAM as a deniability property stating that for every infinitely-long trace of server accesses, there are $(\mathsf{numBlks} - 1)!$ traces of client accesses with identical server observations but different client requests.

$$\forall \pi_0.\ \#\pi_1 : \mathbf{F}\left(\delta_{\pi_j,\pi_k}\right).\ \mathbf{G}\left(\psi_{\pi_0,\pi_1}\right)\ \geq\ (\mathsf{numBlks} - 1)! \tag{6}$$

The binary predicate δ imposes the requirement that the client's request are different in each of the traces captured by the counting quantifier, and the condition in ψ states that all the traces captured by the counting quantifier have the same observable access pattern as π_0.

Verification of Deniability in Path ORAM: To verify the QHP stated in Eq. 6, for every trace of server accesses we need to generate $(\mathsf{numBlks} - 1)!$ traces of client requests that produce the same server access.

Suppose we have Path ORAM (a) that is initialized with some position map. Now consider the Path ORAM (b) with the same number of blocks, but with an initial position map that is a derangement of the position map of (a).[6] The key insight is that ORAM (b) can simulate an identical server access pattern as ORAM (a) by appropriately choosing a different client request that maps to the same leaf that is being accessed by (a) and then updating the position map identically as (a). This is shown in Fig. 7, which shows two Path ORAMs that produce identical server access patterns but service different client requests.

[6] A **derangement** of a set is a permutation of the elements of the set such that no element appears in its original position.

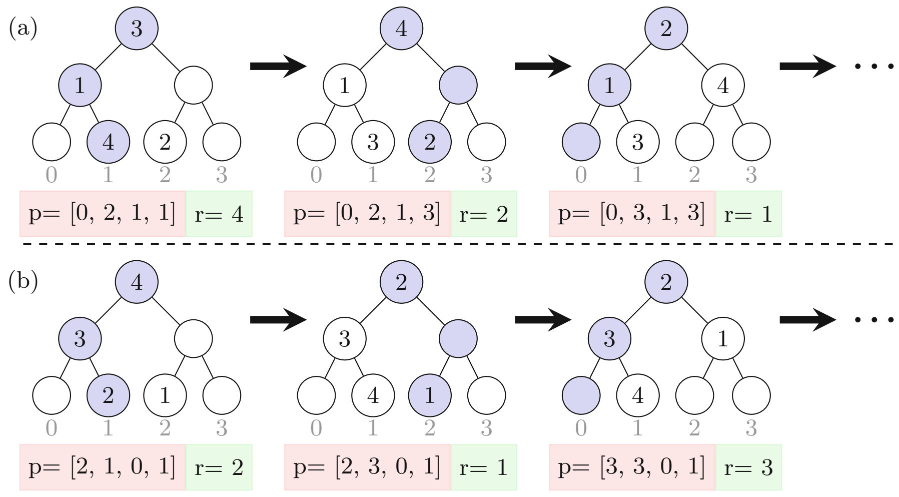


Fig. 7. Path ORAMs satisfying the counting quantifier of Eq. 6, where, p represents the position map indexed from 1 and r is the client's request.

The above insight leads to a trace enumeration where two traces are related via $\mathcal{U}$ if their position maps are derangements of each other, the client accesses are permuted as per the derangement while all other parameters of the ORAM are identical. We use this to prove Property 6. Further details are given in [43].

7 Related Work

Hyperproperties: Research into secure information flow started with the seminal work of Denning and Denning [24], Goguen and Meseguer [32] and Rushby [42]. The self-composition construction for the verification of secure information flow was introduced by Barthe et al. [10]. Clarkson and Schneider [21] introduced the class of specifications called hyperproperties. Clarkson and colleagues also introduced HyperLTL and HyperCTL* [19], which are temporal logics for specifying hyperproperties, while verification algorithms for these were introduced by Finkbeiner and colleagues in [30]. Cartesian Hoare Logic [47] was introduced by Sousa and Dillig and enables the specification and verification of hyperproperties over programs as opposed to transition systems. A number of subsequent efforts have studied hyperproperties in the context of program verification [5,26,45,53].

Quantitative Information Flow: Quantitative hyperproperties build on the rich literature of quantitative information flow (QIF) [3,17,20,34,46]. The QIF problem is to quantify (or bound) the number of bits of secret information that is attacker-observable. Certain notions of QIF can be expressed as QHPs. It is important to note QHPs can express security specifications (e.g., soundness) that are not QIF. Yasuoka and Terauchi studied QIF from a theoretical perspective and showed that it could be expressed as hypersafety and hyperliveness [54]. Approaches based on QIF measures such as min-entropy [46], Shannon entropy [18] etc. have also been applied in the context of static analysis [38].

Quantitative Hyperproperties: Quantitative Cartesian Hoare Logic (QCHL) enables verification of certain quantitative properties of programs [16]. QHPs are more expressive than QCHL, the latter counts events within a trace (e.g. memory accesses), while QHPs count the number of traces satisfying certain conditions.

The most closely related work to ours is of Finkbeiner et al. [29] who introduced Quantitative HyperLTL over Kripke structures. They also introduced a verification algorithm for this logic that is based on maximum model counting. However, their algorithm does not scale to reasonable-sized systems, and experiments from their paper show that the approach times out when checking an 8-bit leak in a password checker (using 8-bit passwords). We differ from their work in three important ways. First, our properties are defined over symbolic transition systems rather than Kripke structures. This allows modeling and verification of QHPs over infinite-state systems. Second, our bounds are symbolic, which enables us to express bounds as functions of transition system parameters. Finally, our definition of Quantitative HyperLTL is also more expressive. It is not possible to convert our QHPs into (non-quantitative) HyperLTL formulas with k-traces for any fixed value of k.

Verification of ORAMs: In concurrent work with ours, Barthe et al. [11] and Darais et al. [22] have introduced specialized mechanisms to prove security of ORAMs. Barthe et al. [11] introduced a probabilistic separation logic (PSL) that (among other things) can be used to reason about the security of ORAMs. Unlike QHPs, PSL does not permit quantitative reasoning about probabilities of events and also does not (yet) support machine-checked reasoning. Darais et al. [22] introduce a type system that enforces obliviousness; they use this type system to implement a tree-based ORAM. Note that QHPs can express specifications other than obliviousness, and obliviousness need not necessarily be a QHP.

8 Conclusion

Quantitative hyperproperties are a powerful class of specifications that stipulate the existence of a certain number of traces satisfying certain constraints. Many important security guarantees, especially those involving probabilistic guarantees of security, can be expressed as quantitative hyperproperties. Unfortunately, verification of quantitative hyperproperties is a challenging problem because these specifications require simultaneous reasoning about a large number of traces of a system. In this paper, we introduced a specification language, satisfaction semantics, and a verification methodology for quantitative hyperproperties. Our verification methodology is based on reducing the problem of counting traces into that of counting the number of assignments that satisfy a first-order logic formula. Our methodology enables security verification of many interesting security protocols that were previously out of reach, including confidentiality of access pattern accesses in Path ORAM.

Acknowledgements. We sincerely thank the anonymous reviewers for their insightful comments, which helped improve this paper. This work was supported in part by the Semiconductor Research Corporation under Task 2854 and the Science and Engineering Research Board of India, a unit of the Department of Science and Technology, Government of India.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F.: Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. In: Peyrin, T. (ed.) FSE 2016. LNCS, vol. 9783, pp. 163–184. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-52993-5_9
2. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security, pp. 53–70 (2016)
3. Alvim, M.S., Andrés, M.E., Palamidessi, C.: Quantitative information flow in interactive systems. J. Comput. Secur. **20**(1), 3–50 (2012)
4. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: PLDI, pp. 362–375 (2017)
5. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, New York, NY, USA, pp. 362–375. ACM (2017)
6. Almeida, J.B., Barbosa, M., Pinto, J.S., Vieira, B.: Formal verification of side-channel countermeasures using self-composition. Sci. Comput. Program. **78**(7), 796–812 (2013)
7. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, SP 2009, Washington, DC, USA, pp. 141–153. IEEE Computer Society (2009)
8. Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1267–1279. ACM (2014)
9. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
10. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop (CSFW-17), pp. 100–114 (2004)
11. Barthe, G., Hsu, J., Liao, K.: A probabilistic separation logic. In: Proceedings of ACM Programming Language, vol. 4, no. POPL, December 2019
12. Bindschaedler, V., Shokri, R., Gunter, C.A.: Plausible deniability for privacy-preserving data synthesis. In: Proceedings of the VLDB Endowment, vol. 10, no. 5, pp. 481–492 (2017)

13. Björner, A., Stanley, R.P.: A Combinatorial Miscellany. L'Enseignement mathématique (2010)
14. Chakraborti, A., Chen, C., Sion, R.: Datalair: efficient block storage with plausible deniability against multi-snapshot adversaries. In: Proceedings on Privacy Enhancing Technologies, vol. 2017, no. 3, pp. 179–197 (2017)
15. Cheang, K., Rasmussen, C., Seshia, S., Subramanyan, P.: A formal approach to secure speculation. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), pp. 288–28815, June 2019
16. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, New York, NY, USA, pp. 875–890. ACM (2017)
17. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. J. Logic Comput. **15**(2), 181–199 (2005)
18. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. J. Comput. Secur. **15**(3), 321–371 (2007)
19. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
20. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Belief in information flow. In: 18th IEEE Computer Security Foundations Workshop (CSFW 2005), pp. 31–45. IEEE (2005)
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
22. Darais, D., Sweet, I., Liu, C., Hicks, M.: A language for probabilistically oblivious computation. In: Proceedings of ACM Programming Language, vol. 4, no. POPL, December 2019
23. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (2008)
24. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
25. Experiments: Models and Proof Scripts for the paper Verification of Quantitative Hyperproperties Using Trace Enumeration Relations (2020). https://github.com/ssahai/CAV-2020-benchmarks
26. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, 15–18 July 2019, Proceedings, Part I, pp. 200–218 (2019)
27. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
28. Ferraiuolo, A., Xu, R., Zhang, D., Myers, A.C., Suh, G.E.: Verification of a practical hardware security architecture through static information flow analysis. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, 8–12 April 2017, pp. 555–568 (2017)
29. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July 2018, Proceedings, Part I, pp. 144–163 (2018)

30. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
31. Fremont, D.J., Rabe, M.N., Seshia, S.A.: Maximum model counting. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)
32. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982, pp. 11–20 (1982)
33. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
34. James, W., Gray, I.I.I.: Toward a mathematical foundation for information flow security. J.Comput. Secur. **1**(3–4), 255–294 (1992)
35. Guarnieri, M., Morales, B.J.F., Reineke, J., Sánchez, A.: SPECTECTOR: principled detection of speculative information flows. CoRR, abs/1812.08639 (2018)
36. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
37. Hawblitzel, C., et al.: Ironclad apps: end-to-end security via automated full-system verification. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, pp. 165–181 (2014)
38. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: International Conference on Computer Aided Verification, pp. 564–580. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_40
39. Muduli, S.K., Subramanyan, P., Ray, S.: Verification of authenticated firmware loaders. In: Proceedings of Formal Methods in Computer-Aided Design. IEEE (2019)
40. Ren, L., et al.: Constants count: practical improvements to oblivious RAM. In: 24th USENIX Security Symposium (USENIX Security 15), Washington, D.C., pp. 415–430, August 2015. USENIX Association (2015)
41. Roscoe, A.W.: CSP and determinism in security modelling. In: Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, 8–10 May 1995, pp. 114–127 (1995)
42. Rushby, J.M.: Proof of separability: a verification technique for a class of a security kernels. In: International Symposium on Programming, 5th Colloquium, Torino, Italy, 6–8 April 1982, Proceedings, pp. 352–367 (1982)
43. Sahai, S., Subramanyan, P., Sinha, R.: Verification of quantitative hyperproperties using trace enumeration relations. arXiv e-prints arXiv:abs/2005.04606, May 2020
44. Seshia, S.A., Subramanyan, P.: Uclid 5: integrating modeling, verification, synthesis and learning. In: Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), October 2018
45. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 161–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_9
46. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_21
47. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, New York, NY, USA, pp. 57–69. ACM (2016)

48. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 299–310 (2013)
49. Subramanyan, P., Sinha, R., Lebedev, I.A., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–03 November 2017, pp. 2435–2450 (2017)
50. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Static Analysis, 12th International Symposium, SAS, Proceedings, pp. 352–367 (2005)
51. UCLID5 Verification and Synthesis System (2019). http://github.com/uclid-org/uclid/
52. Wilf, H.S.: Generatingfunctionology. AK Peters/CRC Press (2005)
53. Yang, W., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, 14–17 July 2018, Proceedings (2018)
54. Yasuoka, H., Terauchi, T.: Quantitative information flow as safety and liveness hyperproperties. Theor. Comput. Sci. **538**, 167–182 (2014)
55. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop, pp. 29–43. IEEE (2003)
56. Zeilberger, D.: Enumerative and algebraic combinatorics. In: The Princeton Companion to Mathematics, pp. 550–561. Princeton University Press (2010)
57. Zhang, D., Wang, Y., Edward Suh, G., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, 14–18 March 2015, pp. 503–516 (2015)

Validation of Abstract Side-Channel Models for Computer Architectures

Hamed Nemati[1], Pablo Buiras[2], Andreas Lindner[2](✉), Roberto Guanciale[2], and Swen Jacobs[1]

[1] Helmholtz Center for Information Security (CISPA), Saarbrücken, Germany
{hnnemati,jacobs}@cispa.saarland

[2] KTH Royal Institute of Technology, Stockholm, Sweden
{buiras,andili,robertog}@kth.se

Abstract. Observational models make tractable the analysis of information flow properties by providing an abstraction of side channels. We introduce a methodology and a tool, Scam-V, to validate observational models for modern computer architectures. We combine symbolic execution, relational analysis, and different program generation techniques to generate experiments and validate the models. An experiment consists of a randomly generated program together with two inputs that are observationally equivalent according to the model under the test. Validation is done by checking indistinguishability of the two inputs on real hardware by executing the program and analyzing the side channel. We have evaluated our framework by validating models that abstract the data-cache side channel of a Raspberry Pi 3 board with a processor implementing the ARMv8-A architecture. Our results show that Scam-V can identify bugs in the implementation of the models and generate test programs which invalidate the models due to hidden microarchitectural behavior.

Keywords: Testing · Side channels · Information flow security · Model validation · Microarchitectures

1 Introduction

Information flow analysis that takes into account side channels is a topic of increasing relevance, as attacks that compromise confidentiality via different microarchitectural features and sophisticated side channels continue to emerge [2,27,28,31–33,40]. While there are information flow analyses that try to counter these threats [3,15], these approaches use models that abstract from many features of modern processors, like caches and pipelining, and their effects on channels that can be accessed by an attacker, like execution time and power consumption. Instead, these models [36] include explicit "observations" that become available to an attacker when the program is executed and that should overapproximate the information that can be observed on the real system.

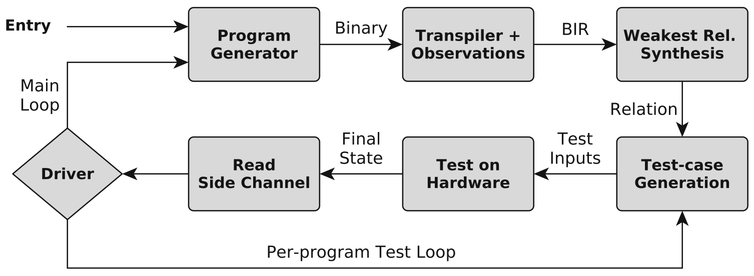


Fig. 1. Validation framework workflow

While abstract models are indispensable for automatic verification because of the complexity of modern microarchitectures, the amount of details hidden by these models makes it hard to trust that no information flow is missed, i.e., their soundness. Different implementations of the same architecture, as well as optimizations such as parallel and speculative execution, can introduce side channels that may be overlooked by the abstract models. This has been demonstrated by the recent Spectre attacks [32]: disregarding these microarchitectural features can lead to consider programs that leak information on modern CPUs as secure. Thus, it is essential to validate whether an abstract model adequately reflects all information flows introduced by the low-level features of a specific processor.

In this work, we introduce an approach that addresses this problem: we show how to validate observational models by comparing their outputs against the behavior of the real hardware in systematically generated experiments. In the following, we give an overview of our approach and this paper.

Our Contribution. We introduce Scam-V (Side Channel Abstract Model Validator), a framework for the automatic validation of abstract observational models. At a high level, Scam-V generates well-formed[1] random binaries and attempts to construct pairs of initial states such that runs of the binaries from these states are indistinguishable at the level of the model, but distinguishable on the real hardware. In essence, finding such counterexamples implies that the observational model is not sound, and leads to a potential vulnerability. Figure 1 illustrates the main workflow of Scam-V.

The first step of our workflow (described in Sect. 3) is the generation of a binary program for the given architecture, guided towards programs that trigger certain features of the architecture. The second step translates the program to the intermediate language BIR (described in Sect. 2.4) and annotates the result with observations according to the observational model under validation. This transpilation is provably correct with respect to the formal model of the ISA, i.e., the original binary program and the transpiled BIR program have the same effects on registers and memory. In step three we use symbolic execution to syn-

[1] Terminating programs which do not cause run-time exceptions and emit observations required by the analysis.

thesize the weakest relation on program states that guarantees indistinguishability in the observational model (Sect. 4). Through this relation, the observational model is used to drive the generation of *test cases* – pairs of states that satisfy the relation and can be used as inputs to the program (Sect. 5). Finally, we run the generated binary with different test cases on the real hardware, and compare the measurements on the side channel of the real processor. A description of this process together with general remarks on our framework implementation are in Sect. 6. Since the generated test cases satisfy the synthesized relation, soundness of the model would imply that the side-channel data on the real hardware cannot be distinguished either. Thus, a test case where we can distinguish the two runs on the hardware amounts to a counterexample that invalidates the observational model. After examining a given test case, the driver of the framework decides whether to generate more test cases for the same program, or to generate a new program.

We have implemented Scam-V in the HOL4 theorem prover[2] and have evaluated the framework on three observational models (introduced in Sect. 2.3) for the L1 data-cache of the ARMv8 processor on the Raspberry Pi 3 (Sect. 2.2). Our experiments (Sect. 7) led to the identification of model invalidating microarchitectural features as well as bugs in the ARMv8 ISA model and our observational extensions. This shows that many existing abstractions are substantially unsound.

Since our goal is to validate that observational models overapproximate hardware information flows, we do not attempt to identify practically exploitable vulnerabilities. Instead, our experiments attempt to validate these models in the worst case scenario for the victim. This consists of an attacker that can precisely identify the cache lines that have been evicted by the victim and that can minimize the noise of these measurements in the presence of background processes and interrupts.

2 Background

2.1 Observational Models

We briefly introduce the concepts of side channels, indistinguishability, observational models, and observational equivalence. For the rest of this section, consider a fixed program that runs on a fixed processor. We can model the program running on the processor by a transition system $M = \langle S, \rightarrow \rangle$, where S is a set of states and $\rightarrow \subseteq S \times S$ a transition relation. In automated verification, the state space of such a model usually reflects the possible values of program variables (or: registers of the processor), abstracting from low-level behavior of the processor, such as cache contents, electric currents, or real-time behavior. That is, for every state of the real system there is a state in the model that represents it, and a state of the model usually represents a set of states of the real system.

Then, a *side channel* is a trait of the real system that can be read from by an attacker and that is not modeled in M.

[2] https://hol-theorem-prover.org.

Definition 1 (Indistinguishability). *States r_1 and r_2 of the real system are* indistinguishable *if a real-world attacker is not able to distinguish executions from r_1 or r_2 by means of the side channel on the real hardware.*

Note that executions may be distinguishable even if they end in the same final state, e.g., if the attacker is able to measure execution time.

In order to verify resilience against attacks that use side channels, one option is to extend the model to include additional features of the real system and to formalize indistinguishability in terms of some variations of non-interference [25,26]. Unfortunately, it is infeasible to develop formal models that capture *all* side channels of a modern computer architecture. For instance, precisely determining execution time or power consumption of a program requires to deal with complex processor features such as cache hierarchies, cache replacement policies, speculative execution, branch prediction, or bus arbitration. Moreover, for some important parts of microarchitectures, their exact behavior may not even be public knowledge, e.g., the mechanism used to train the branch predictor. Additionally, information flow analyses cannot use the same types of overapproximations that are used for checking safety properties or analyzing worst-case execution time, e.g., the introduction of nondeterminism to cover all possible outcomes.

In order to handle this complexity, information flow analyses [3,15] use models designed to overapproximate information flow to channels in terms of system state observations. To this end, the model is extended with a set of possible observations O and we consider a transition relation $\rightarrow \subseteq S \times O \times S$, i.e., each transition produces an observation that captures the information that it potentially leaks to the attacker. We assume that the set O contains an *empty observation* $\perp$, and call a transition labeled with $\perp$ a *silent* transition. We call the resulting transition system an *observational model.* For instance, in case of a rudimentary cacheless processor, the execution time of a program depends only on the sequence of executed instructions. In this case, extending the model with observations that reveal the instructions is more convenient than producing a clock-accurate model of the system.

We use the operator $\circ$ for the sequential composition of observations. In particular, for a trace $\pi = s_0 \rightarrow^{o_1} s_1 \ldots \rightarrow^{o_n} s_n$ of the model, we write $o_1 \circ \ldots \circ o_n$ for the sequence of observations along π. We write $o_1 \circ \ldots \circ o_n \approx o'_1 \circ \ldots \circ o'_{n'}$ if the two sequences are equal after removing silent transitions. Comparing traces with observations leads to a notion of *observational equivalence*, defined as a relation on program states.

Definition 2 (Observational equivalence). *Traces $\pi = s_0 \rightarrow^{o_1} s_1 \ldots \rightarrow^{o_n} s_n$ and $\pi' = s'_0 \rightarrow^{o'_1} s'_1 \ldots \rightarrow^{o'_{n'}} s'_{n'}$ of an observational model M are* observationally equivalent *(written as $\pi \sim_M \pi'$) iff $o_1 \circ \ldots \circ o_n \approx o'_1 \circ \ldots \circ o'_{n'}$.*

States $s_1 \in S$ and $s_2 \in S$ are observationally equivalent, *denoted $s_1 \sim_M s_2$, iff for every possible trace π_1 of M that starts in s_1 there is a trace π_2 of M that starts in s_2 such that $\pi_1 \sim_M \pi_2$, and vice versa.*

Note that this notion is, in principle, different from the notion of *indistinguishability.* The overapproximation of information flows can lead to false positives: for example, execution of a program may require the same amount of time

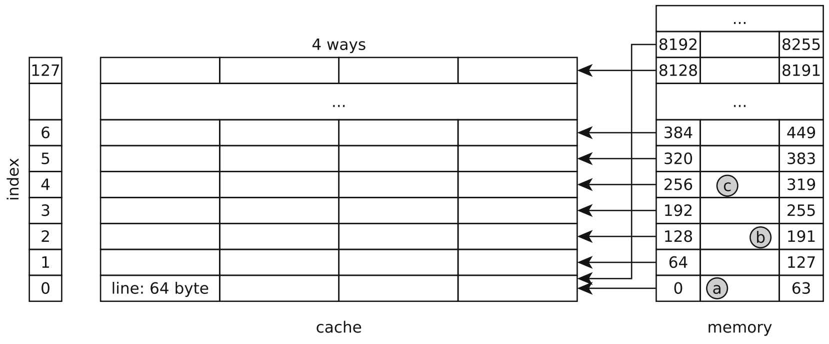


Fig. 2. L1 data-cache structure.

even if the sequences of executed instructions are different. A more severe concern is that these abstractions may overlook some flows of information due to the number of low-level details that are hidden. For instance, an observational model may not take into account that for some microcontrollers the number of clock cycles required for multiplication depends on the value of the operands.

The use of an abstract model to verify resilience against side-channel attacks relies on the assumption that observational equivalence entails indistinguishability for a real-world attacker on the real system:

Definition 3 (Soundness). *An observational model M is* sound *if whenever the model states s_1 and s_2 represent the real system states r_1 and r_2, respectively, then $s_1 \sim_M s_2$ entails indistinguishability of r_1 and r_2.*

2.2 The Evaluation Platform: Raspberry Pi 3

In order to evaluate our framework, we selected Raspberry Pi 3[3], which is a widely available ARMv8 embedded system. The platform's CPU is a Cortex-A53, which is an 8-stage pipelined processor with a 2-way superscalar and in-order execution pipeline. The CPU implements branch prediction, but it does not support speculative execution. This makes the CPU resilient against variations of Spectre attacks [5].

In the following, we focus on side channels that exploit the Level 1 (L1) data-cache of the system. The L1 data-cache is transparent for programmers. When the CPU needs to read a location in memory in case of a cache miss, it copies the data from memory into the cache for subsequent uses, tagging it with the memory location from which the data was read.

Data is transferred between memory and cache in blocks of 64 bytes, called cache lines. The L1 data-cache (Fig. 2) is physically indexed and physically tagged and is 4-way set associative: each memory location can be cached in four different entries in the cache—when a line is loaded, if all corresponding entries

[3] https://www.raspberrypi.org.

are occupied, the CPU uses a specific (and usually underspecified) replacement policy to decide which colliding line should be evicted. The whole L1 cache is 32KB in size, hence it has 128 cache sets (i.e. 32 KB/64 B/4). Let a be a physical address, in the following we use $\texttt{off}(a)$ (i.e., least significant 6 bits), $\texttt{index}(a)$ (i.e., bits from 6 to 12), and $\texttt{tag}(a)$ (i.e., the remaining bits) to extract the cache offset, cache set index, and cache tag of the address.

The cache implements a prefetcher, for some configurable $k \in \mathbb{N}$: when it detects a sequence of k cache misses whose cache set indices are separated by a fixed stride, the prefetcher starts to fetch data in the background. For example, in Fig. 2, if $k = 3$ and the cache is initially empty then accessing addresses a, b, and c, whose cache lines are separated by a stride of 2, can cause the cache to prefetch the block $[384 \dots 449]$.

2.3 Different Attacker and Observational Models

Attacks that exploit the L1 data-cache are usually classified in three categories: In *time-driven attacks* (e.g. [47]), the attacker measures the execution time of the victim and uses this knowledge to estimate the number of cache misses and hits of the victim; In *trace-driven attacks* (e.g. [1,48]), the adversary can profile the cache activities during the execution of the victim and observe the cache effects of a particular operation performed by the victim; Finally, in *access-driven attacks* (e.g. [39,46]), the attacker can only determine the cache sets modified after the execution of the victim has completed. A widely used approach to extract information via cache is Prime+Probe [40]: (1) the attacker reads its own memory, filling the cache with its data; (2) the victim is executed; (3) the attacker measures the time needed to access the data loaded at step (1): slow access means that the corresponding cache line has been evicted in step (2).

In the following we disregard time-driven attacks and trace-driven attacks: the former can be countered by normalizing the victim execution time; the latter can be countered by preventing victim preemption. Focusing on access-driven attacks leads to the following notion of indistinguishability:

Definition 4. *Real system states r_1 and r_2 are* indistinguishable for access-driven attacks on the L1 data-cache *iff executions starting in r_1 or r_2 modify the same cache sets.*

We remark that for multi-way caches, the need for models that overapproximate the information flow is critical since the replacement policies are seldom formally specified and a precise model of the channel is not possible. The following observational model attempts to overapproximate information flows for data-caches by relying on the fact that accessing two different addresses that only differ in their cache offset produces the same cache effects:

Definition 5. *The transition relation of the* multi-way cache and pc observational model *is $s \to^{o}_{mwc,pc} s'$, where $\to^{o}_{mwc,pc}$ models the execution of one single instruction, with $o \in \mathbb{N} \times ((\{rd, wt\} \times \mathbb{N} \times \mathbb{N}) \cup \bot)$. If $o = (pc, acc)$ then pc is the current program counter and $acc = (op, t, i)$ is the memory access performed* *by the instruction, where op is the memory operation, t is the cache tag and i is the cache set index corresponding to the address. If the instruction does not access the memory, then $acc = \perp$.*

Notice that by making the program counter observable, this model assumes that the attacker can infer the sequence of instructions executed by the program.

We introduce several relaxed models, representing different assumptions on the hardware behavior and attacker capability. Each relaxed model is obtained by projecting observations of Definition 5. Let α be a relaxed model and f_α the corresponding projection function, then $s \rightarrow_{\alpha}^{o'} s'$ iff exists o such that $f_\alpha(o) = o'$ and $s \rightarrow_{mwc,pc}^{o} s'$.

The following model assumes that the effects of instructions that do not interact with the data memory are not measurable, hence the attacker does not observe the program counter:

Definition 6. *The projection of the* multi-way cache observational model *is $f_{mwc}((pc, acc)) = acc$.*

On many processors, the replacement policy for a cache set does not depend on previous accesses performed to other cache sets. The resulting isolation among cache sets leads to the development of an efficient countermeasure against access-driven attacks: cache coloring [23,45]. This consists in partitioning the cache sets into multiple regions and ensuring that memory pages accessible by the adversary are mapped to a specific region of the cache. In this case, accesses to other regions do not affect the state of cache sets that an attacker can examine. Therefore these accesses are not observable. This assumption is captured by the following model:

Definition 7. *The projection of the* partitioned multi-way cache observational model *is $f_{pmwc}((pc, acc)) = acc$ if $acc = (op, t, i)$ and i belongs to the set of cache sets that are addressable by the attacker, and is $\perp$ otherwise.*

Notice that cache prefetching can violate soundness of this model, since accesses to the non-observable region of the cache may lead to prefetching addresses that lie in the observable part of the cache (see Sect. 7.2).

Finally, for direct-mapped caches, where each memory address is mapped to only one cache entry, the cache tag should not be observable if the attacker does not share memory with the victim:

Definition 8. *The projection of the* direct-mapped cache observational model *is $f_{dc}((pc, (op, t, i))) = (op, i)$ and $f_{dc}((pc, \perp)) = \perp$.*

Since the cache in Cortex-A53 is multi-way set associative, this model is not sound. For example, in a two-way set associative cache, accessing a, a and a, b, where both a and b have the same cache set index but different cache tags, may result in different cache states.

BIR program with observation

```
//b.eq l2
[l0:CJMP Z l2 l1]
//mul x1 x2 x3
[l1:X1= X2*X3; JMP l2]
//ldr x2 {x1} +8
[l2:OBS(sline(X1),[tag(X1),index(X1)]);
    X2= LOAD(M, X1);X1= X1+8;HALT]
```

Fig. 3. BIR transpilation example

2.4 Binary Intermediate Representation

To achieve a degree of hardware independence, we use the architecture-agnostic intermediate representation BIR [34]. It is an abstract assembly language with statements that work on memory, arithmetic expressions, and jumps. Figure 3 shows an example of code in a generic assembly language and its transpiled BIR code. This code performs a conditional jump to *l2* if **Z** holds, and otherwise it sets **X1** to the multiplication **X2** $*$ **X3**. Then, at *l2* it loads a word from memory at address **X1** into **X2**, and finally adds 8 to the pointer **X1**. BIR programs are organized into blocks, which consist of jump-free statements and end in either conditional jump (`CJMP`), unconditional jump (`JMP`), or `HALT`.

BIR also has explicit support for *observations*, which are produced by statements that evaluate a list of expressions in the current state. To account for expressive observational models, BIR allows conditional observation. The condition is represented by an expression attached to the observation statement. The observation itself happens only if this condition evaluates as true in the current state. The observations in Fig. 3 reflect a scenario where the data-cache has been partitioned: some lines are exclusively accessible by the victim (i.e. the program), some lines can be shared with the attacker. The statement `OBS(sline(`**X1**`), [tag(`**X1**`), index(`**X1**`)])` for the load instruction consists of an observation condition (`sline(`**X1**`)`) and a list of expressions to observe (`[tag(`**X1**`), index(`**X1**`)]`). The function `sline` checks that the argument address is mapped in a shared line and therefore visible to the attacker. The functions `tag` and `index` extract the cache tag and set index in which the argument address is mapped. Binary programs can be translated to BIR via a process called *transpilation*. This transformation reuses formal models of the ISAs and generates a proof that certifies correctness of the translation by establishing a bisimulation between the two programs.

3 Program Generation

We base our validation of observational models on the execution of binary programs rather than higher-level code representations. This approach has the following benefits: (i) It obviates the necessity to trust compilers or reason about

```
Random program generator
udiv x3, x16, x8
cbnz x28, #12
bics x14, x3, x26, ror #21
ldrsb w4, [x24, x3]
ldp x22, x14, [x3], #0xD0
```

Fig. 4. Example programs generated by the Scam-V random program generator.

```
  Load generator         Load strides            Load seq. with branches
ldr x5, [x1, 4]  | ldr x16, [x2, #0]     | cmp x13, x3
ldr x0, x3       | ldr x19, [x2, #64]    | b.eq #0x14
ldr x2, [x9, 8]  | ldr x1,  [x2, #128]   | ldr x6, #0x1DFA
                 | ldr x22, [x2, #192]   | ldr x9, [x20]
                 | ldr x17, [x2, #256]   | ldr x20, [x22, #8]
                 |                       | b #0x10
                 |                       | ldr x16, [x0, #16]
                 |                       | ldr x16, #0x1BE8
                 |                       | ldr x12, #0x17D5
```

Fig. 5. Example programs generated by Scam-V monadic program generators.

how their compilation affects side-channels. (ii) Implementation effort is reduced because most existing side-channel analysis approaches also operate on binary representations, which requires ISA models. (iii) This approach allows to find ISA model faults independently of the compilation. (iv) It enables a unified infrastructure to handle many different types of channels.

In Scam-V, we implemented two techniques to generate well-formed binaries: *random* program generation and *monadic* program generation. The random generator leverages the instruction encoding machinery from the existing HOL4 model of the ISA and produces arbitrary well-formed ARMv8 binaries, with the possibility to control the frequency of occurrences of each instruction class. The monadic generator is following a grammar-driven approach in the style of QuickCheck [13] that generates arbitrary programs that fit a specific pattern or template. The program templates can be defined in a modular, declarative style and are extensible. We use this approach to generate programs in a guided fashion, focusing on processor features that we want to exercise in order to validate a model, or those we suspect may lead to a counterexample. Figures 4 and 5 show some example programs generated by Scam-V, including straight-line programs that only do memory loads, programs that load from addresses in a stride pattern to trigger automatic prefetching, and programs with branches. More details on how the program generators work can be found in [38].

4 Synthesis of Weakest Relation

Synthesis of the weakest relation is based on standard symbolic execution techniques. We only cover the basic ideas of symbolic execution in the following and refer the reader to [30] for more details. We use $\mathbf{X}$ to range over symbols, and $\mathbf{c}$, $\mathbf{e}$, and $\mathbf{p}$ to range over symbolic expressions. A symbolic state σ consists of a concrete program counter i_σ, a path condition $\mathbf{p}_\sigma$, and a mapping $\mathbf{m}_\sigma$ from variables to symbolic expressions. We write $e(\sigma) = \mathbf{e}$ for the symbolic evaluation of the expression e in σ, and $\mathbf{e}(s)$ for the value obtained by substituting the symbols of the symbolic expression $\mathbf{e}$ with the values of the variables in s, where s is a concrete state.

Symbolic execution produces one terminating state[4] for each possible execution path: a terminating state is produced when `HALT` is encountered; the execution of `CJMP` $c\ l_1\ l_2$ from state σ follows both branches using the path conditions $c(\sigma)$ and $\neg c(\sigma)$. Symbolic execution of the example in Fig. 3 produces the terminating states σ_1 and σ_2. For the first branch we have $\mathbf{p}_{\sigma_1} = \mathbf{Z}$ and $\mathbf{m}_{\sigma_1} = \{X_1 \to \mathbf{X}_1 + 8, X_2 \to \texttt{LOAD}(\mathbf{M}, \mathbf{X}_1)\}$ (we omit the variables that are not updated), and for the second branch $\mathbf{p}_{\sigma_2} = \neg\mathbf{Z}$ and $\mathbf{m}_{\sigma_2} = \{X_1 \to \mathbf{X}_2 * \mathbf{X}_3 + 8, X_2 \to \texttt{LOAD}(\mathbf{M}, \mathbf{X}_2 * \mathbf{X}_3)\}$.

We extend standard symbolic execution to handle observations. That is, we add to each symbolic state a list $\mathbf{l}_\sigma$, and the execution of `OBS` $c\ \vec{e}$ in σ appends the pair $(\mathbf{c}, \vec{\mathbf{e}})$ to $\mathbf{l}_\sigma$, where $\mathbf{c} = c(\sigma)$ and $\vec{\mathbf{e}}[i] = \vec{e}[i](\sigma)$ are the symbolic evaluation of the condition and expressions of the observation. For instance, in the example of Fig. 3 the list for the terminating states are

$$\begin{aligned}\mathbf{l}_{\sigma_1} &= [(\texttt{sline}(\mathbf{X}_1), [\texttt{tag}(\mathbf{X}_1), \texttt{index}(\mathbf{X}_1)])]\\ \mathbf{l}_{\sigma_2} &= [(\texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3), [\texttt{tag}(\mathbf{X}_2 * \mathbf{X}_3), \texttt{index}(\mathbf{X}_2 * \mathbf{X}_3)])]\end{aligned}$$

Let Σ be the set of terminating states produced by the symbolic execution, s be a concrete state, and $\sigma \in \Sigma$ be a symbolic state such that $\mathbf{p}_\sigma(s)$ holds, then executing the program from the initial state s produces the value $\mathbf{m}_\sigma(X)(s)$ for the variable X. Moreover, let $\mathbf{l}_\sigma = [(\mathbf{c}_1, \vec{\mathbf{e}}_1) \ldots (\mathbf{c}_n, \vec{\mathbf{e}}_n)]$, then the generated observations are $(\mathbf{c}_1, \vec{\mathbf{e}}_1)(s) \circ \ldots \circ (\mathbf{c}_n, \vec{\mathbf{e}}_n)(s)$, where $(\mathbf{c}_1, \vec{\mathbf{e}}_1)(s) = \vec{\mathbf{e}}_1(s)$ if $\mathbf{c}_1(s)$, and otherwise $\perp$ (i.e. observations are list of concrete values).

After computing Σ, we synthesize the observational equivalence relation (denoted by $\sim$) by ensuring that every possible pair of execution paths have equivalent lists of observations. Formally, $s_1 \sim s_2$ is equivalent to:

$$\bigwedge_{(\sigma_1, \sigma_2) \in \Sigma \times \Sigma} (\mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2) \Rightarrow \mathbf{l}_{\sigma_1}(s_1) = \mathbf{l}_{\sigma_2}(s_2))$$

This synthesized relation implies the observational equivalence defined in Sect. 2 (Definition 2). In the example, the synthesized relation (after simplification) is as follows (notice that primed symbols represent variables of the second state and we omitted the symmetric cases):

[4] We consider only terminating programs.

$$\overset{s_1 =}{\left\{\begin{array}{l}\mathbf{Z} = T\\ \mathbf{X}_1 = 130\\ \mathbf{X}_2 = 123546\\ \mathbf{X}_3 = 87465\end{array}\right\}} \sim \overset{s_2 =}{\left\{\begin{array}{l}\mathbf{Z} = F\\ \mathbf{X}_1 = 37846\\ \mathbf{X}_2 = 2\\ \mathbf{X}_3 = 64\end{array}\right\}} \quad \overset{s'_1 =}{\left\{\begin{array}{l}\mathbf{Z} = F\\ \mathbf{X}_1 = 3246\\ \mathbf{X}_2 = 64\\ \mathbf{X}_3 = 30\end{array}\right\}} \sim \overset{s'_2 =}{\left\{\begin{array}{l}\mathbf{Z} = F\\ \mathbf{X}_1 = 856\\ \mathbf{X}_2 = 12\\ \mathbf{X}_3 = 64\end{array}\right\}}$$

Fig. 6. Example test cases when the first 10 cache sets are shared.

$$\begin{array}{l}
(\mathbf{Z} \wedge \mathbf{Z}') \Rightarrow \\
\quad \left(\begin{array}{l}\texttt{sline}(\mathbf{X}_1) = \texttt{sline}(\mathbf{X'}_1) \wedge \\ \texttt{sline}(\mathbf{X}_1) \Rightarrow (\texttt{tag}(\mathbf{X}_1) = \texttt{tag}(\mathbf{X'}_1) \wedge \texttt{index}(\mathbf{X}_1) = \texttt{index}(\mathbf{X'}_1))\end{array}\right) \wedge \\
(\mathbf{Z} \wedge \neg\mathbf{Z}') \Rightarrow \\
\quad \left(\begin{array}{l}\texttt{sline}(\mathbf{X}_1) = \texttt{sline}(\mathbf{X'}_2 * \mathbf{X'}_3) \wedge \\ \texttt{sline}(\mathbf{X}_1) \Rightarrow (\texttt{tag}(\mathbf{X}_1) = \texttt{tag}(\mathbf{X'}_2 * \mathbf{X'}_3) \wedge \texttt{index}(\mathbf{X}_1) = \texttt{index}(\mathbf{X'}_2 * \mathbf{X'}_3))\end{array}\right) \wedge \\
(\neg\mathbf{Z} \wedge \neg\mathbf{Z}') \Rightarrow \\
\quad \left(\begin{array}{l}\texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3) = \texttt{sline}(\mathbf{X'}_2 * \mathbf{X'}_3) \wedge \\ \texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3) \Rightarrow (\texttt{tag}(\mathbf{X}_2 * \mathbf{X}_3) = \texttt{tag}(\mathbf{X'}_2 * \mathbf{X'}_3) \wedge \texttt{index}(\mathbf{X}_2 * \mathbf{X}_3) = \texttt{index}(\mathbf{X'}_2 * \mathbf{X'}_3))\end{array}\right)
\end{array}$$

We recall that Raspberry Pi 3 has 128 cache sets and 64 bytes per line. Figure 6 shows two pairs of states that satisfy the relation, assuming only the first 10 cache sets are shared. States s_1 and s_2 lead the program to access the third cache set, while s'_1 and s'_2 lead the program to access cache sets that are not shared, therefore they generate no observations.

5 Test-Case Generation

A test case for a program P is a pair of initial states s_1, s_2 such that P produces the same observations when executed from either state, i.e., $s_1 \sim s_2$. The relation as described in Sect. 4 characterizes the space of observationally equivalent states, so a simple but naive approach to test-case generation consists in querying the SMT solver for a model of this relation. The model that results from the query gives us two concrete observationally equivalent values for the registers that affect the observations of the program, so at this point we could forward these to our testing infrastructure to perform the experiment on the hardware.

However, the size of an observational equivalence class can be enormous, because there are many variations to the initial states that cannot have effects on the channels available to the attacker. Choosing a satisfying assignment for the entire relation every time without any extra guidance risks producing many test cases that are too similar to each other, and thus unlikely to find counterexamples. For instance, the SMT solver may generate many variations of the test case (s_1, s_2) in Fig. 6 by iterating over all possible values for register X_2 of state s_1, even if the value of this register is immaterial for the observation.

In practice, we explore the space of observationally equivalent states in a more systematic manner. To this end, Scam-V supports two mechanisms to guide the selection of test cases: *path enumeration* and *term enumeration*. Path enumeration partitions the space according to the combination of symbolic execution paths that are taken, whereas term enumeration partitions the space according to the value of a user-supplied BIR expression. In both cases, the partitions are explored in round-robin fashion, choosing one test case from each partition in turn. To make the queries to the SMT solver more efficient, we only generate a fragment of the relation that corresponds to the partition under test.

Path Enumeration. Every time we have to generate a test case, we first select a pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ of symbolic states as per Sect. 4, which identifies a pair of paths $(\mathbf{p}_{\sigma_1}, \mathbf{p}_{\sigma_2})$. The chosen paths vary in each iteration in order to achieve full path coverage. The query given to the SMT solver then becomes[5]

$$\mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2) \wedge \mathbf{l}_{\sigma_1}(s_1) = \mathbf{l}_{\sigma_2}(s_2)$$

Since the meat of the relation is a conjunction of implications, this is a natural partitioning scheme that ensures all conjuncts are actually explored. Note that without this mechanism, the SMT solver could always choose states that only satisfy one and the same conjunct. To guide this process even further, the user can supply a *path guard*, which is a predicate on the space of paths. Any path not satisfying the guard is skipped, allowing the user to avoid exploring unwanted paths. For example, for the program in Fig. 3 we can use a path guard to force the test generation to select only paths that produce no observations: e.g., $(\mathbf{Z} \Rightarrow \neg\texttt{sline}(\mathbf{X}_1)) \wedge (\neg\mathbf{Z} \Rightarrow \neg\texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3))$.

Term Enumeration. In addition to path enumeration, we can choose a BIR expression e that depends on the symbolic state, and a range R of values to enumerate. Every query also includes the conjuncts $e_{\sigma_1} = v_1 \wedge e_{\sigma_2} = v_2$ where $v_1, v_2 \in R$ and such that the v_i are chosen to achieve full coverage of $R \times R$. Term enumeration can be useful to introduce domain-specific partitions, provided that $R \times R$ is small enough. For example, this mechanism can be used to ensure that we explore addresses that cover all possible cache sets, if we set e to be a mask that extracts the cache set index bits of the address. For example, for the program in Fig. 3 we can use $\mathbf{Z} * \texttt{index}(\mathbf{X}_1) + (1 - \mathbf{Z}) * \texttt{index}(\mathbf{X}_2 * \mathbf{X}_3)$ to enumerate all combinations of accessed cache sets while respecting the paths.

6 Implementation

The implementation[6] of Scam-V is done in the HOL4 theorem prover using its meta-language, i.e., SML. Scam-V relies on the binary analysis platform HolBA for transpiling the binary code of test programs to the BIR representation. This

[5] Note that this is equivalent to taking a fragment of the observational equivalence relation, specifically the case when $\mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2)$ holds.

[6] Our implementation of Scam-V is embedded in HolBA, which is available at https://github.com/kth-step/HolBA. Our extendable experimentation platform consists of several "EmbExp-*" repositories available at https://github.com/kth-step.

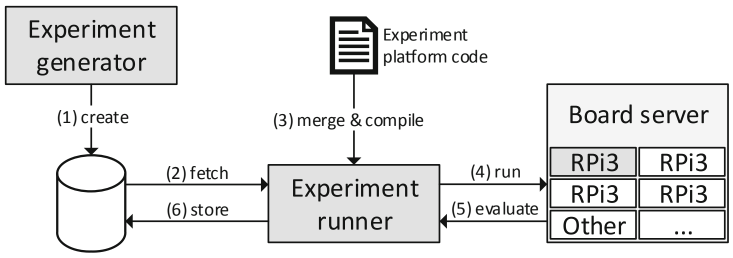


Fig. 7. Experiment handling design with numbered steps. This showcases the workflow for producing, preparing, executing and evaluating one experiment.

transpilation uses the existing HOL4 model of the ARMv8 architecture [16] for giving semantics to ARM programs. In order to validate the observational models of Sect. 2.3, we extended the transpilation process to inline observation statements into the resulting BIR program. These observations represent the observational power of the side channel. In order to compute possible execution paths of test programs and their corresponding observations, which are needed to synthesize the observational equivalence relation of Sect. 4, we implemented a symbolic execution engine in HOL4. All program generators from Sect. 3 as well as the weakest relation synthesis from Sect. 4 and the test-case generator from Sect. 5 are implemented as SML libraries in Scam-V. The latter uses the SMT solver Z3 [14] to generate test inputs. For conducting the experiments in this paper, we used Raspberry Pi 3 boards equipped with ARM Cortex-A53 processors implementing the ARMv8-A architecture.

The Scam-V pipeline generates programs and pairs of observationally equivalent initial states (test cases) for each program. Each combination of a program with one of its test cases is called an *experiment*. After generating experiments, we execute them on the processor implementation of interest to examine their effects on the side channel. Figure 7 depicts the life of a single experiment as goes through our experiment handling design. This consists of: (step 1) generating an experiment and storing it in a database, (step 2) retrieving the experiment from the database, (step 3) integrating it with *experiment-platform code* and compiling it into platform-compatible machine code, and (step 4–6) executing the generated binary on the real board, as well as finally receiving and storing the experiment result.

The experiment-platform code configures page tables to setup *cacheable* and *uncacheable* memory, clears the cache before every execution of the program, and inserts memory barriers around the experiment code. The platform executes in ARM TrustZone, which enables us to use privileged debug instructions to obtain the cache state directly for comparison after experiment execution.

The way in which we compare final cache states for distinguishability depends on the attacker and observational model in question. For multi-way cache, we say two states are *indistinguishable* if and only if for each valid entry in one state, there is a valid entry with the same cache tag in the corresponding cache set of the other state and vice versa. For the partitioned multi-way cache, we check the states in the same way, except we do it only for a subset of the cache sets (see Sect. 7.2 for details on the exact partition). For the direct-mapped cache, we compare how many valid cache lines there are in each set, disregarding the cache tags. These comparison functions have been chosen to match the attacker power of the relaxed models in Definitions 6, 7, and 8 respectively.

7 Results

Since the ARM-v8 experimentation platform runs as bare-metal code, there are no background processes or interrupts. Despite this fact, our measurements may contain noise due to other hardware components that share the same memory subsystem, such as the GPU, and because our experiments are not synchronized with the memory controller. In order to simplify repeatability of our experiments, we execute each experiment 10 times and check for discrepancies in the final state of the data cache. Unless all executions give the same result, this experiment is classified as *inconclusive* and excluded from further analysis.

7.1 Direct-Mapped Cache Observational Model

First, we want to make sure that Scam-V can invalidate unsound observational models in general. For this purpose, we generated experiments that use the model of Definition 8, i.e., for every memory access in BIR we observe the cache set index of the address of the access. We know that this is not a sound model for Raspberry Pi 3, because the platform uses a 4-way cache. Table 1.1 shows that both the random program generator and the monadic load generator uncovered counterexamples that invalidated this observational model.

7.2 Partitioned Cache Observational Model

Next, we consider the partitioned cache observational model from Definition 7. That is, we partition the L1 cache of the Raspberry Pi 3 into two contiguous regions and assume that the attacker has only access to the second region. Due to the prefetcher of Cortex-A53 we expect this model to be unsound and indeed we could invalidate it.

To this end, we generated experiments for two variations of the model. Variation A splits the cache at cache set 61, meaning that only cache sets 61–127 were considered accessible to the attacker. Variation B splits the cache at cache set 64 (the midpoint), such that cache sets 64–127 were considered visible. The following program is one of the counterexamples for variation A that have been discovered by Scam-V using the monadic program generator.

```
Program                              Input 1       Input 2

ldr x2,  [x10, #0]      x10:  0x80100080    0x80100cc0
ldr x20, [x10, #128]
ldr x17, [x10, #256]
```

Table 1. Invalidation of cache and faulty observational models.

(1.1)	Observations	Cache set index only (Definition 8)		
	Programs	Monadic load generator	Random program generator	
	Experiments	39660	20872	
	- Inconclusive	0	1	
	- Counterexample	19	18	
(1.2)	Experiment set	Variation A	Variation B	
	Observations	Page unaligned cache partitioning (Definition 7)	Page aligned cache partitioning (Definition 7)	
	Programs	Monadic stride generator		
	Experiments	36160	37843	
	- Inconclusive	5426	6967	
	- Counterexample	3460	0	
(1.3)	Observations	Cache tag and set index (Definition 6)		
	Programs	Random program generator	Monadic generator	
			Loads	Previction
	Experiments	20256	23120	23290
	- Inconclusive	2	0	0
	- Counterexample	0	5	16
(1.4)	Observations	Cache tag and set index (Definition 6)		
	Programs	Random program generator		
	Experiments	22321		
	- Inconclusive	0		
	- Failure	308		

The counterexample exploits the fact that prefetching fills more lines than those loaded by the program, provided the memory accesses happen in a certain stride pattern. Thus, it essentially needs to have two properties: (i) two different starting addresses for the stride, a_1 and a_2, with a cache set index that is lower than 61 to avoid any observations in the model, and thus satisfying observational equivalence, and (ii) one of a_1 and a_2 is close enough to the partition boundary. In this case, automatic prefetching will continue to fill lines in subsequent sets, effectively crossing the boundary into the attacker-visible region.

In our experiments, we used a path guard to generate only states that produce only memory accesses to the region of the cache that is not visible by the attacker. Additionally, we used term enumeration to force successive test cases to start a stride on a different cache set and therefore cover the different cache set indices. Without this guidance, the tool could generate only experiments that affect the lower sets of the cache and never explore scenarios that affect the sets with indices closer to the split boundary.

For variation B, we have not found such a counterexample. The only difference is that the partition boundary is on line 64, which means that each partition fits exactly in a small page (4K). We conjecture that the prefetcher does not perform line fills across small page (4K) boundaries. This could be for performance reasons, as crossing a page boundary can involve a costly page walk if the next page is not in the TLB. If this is the case, it would seem that it is safe to use prefetching with a partitioned cache, provided the partitions are page-aligned. Table 1.2 summarizes our experiments for this model.

7.3 Multi-way Cache Observational Model

In the remaining experiments, we consider the model of Definition 6 and we assume that the attacker has access to the complete L1 cache. Even if we expected this model to be sound, our experiments (Table 1.3) identified several counterexamples. We comment on two classes of counterexamples below.

Prevction. Some counterexamples are due to an undocumented behavior that we called "prevction" because it causes a cache line to be evicted before the corresponding cache set is full. The following program compares `x0` and `x1` and executes a sequence of three loads. In case of equality, fourteen `nop` are executed between the first two loads.

```
   Program                          Input 1        Input 2

cmp x0, x1                     x0: 0x00000000   0x00000000
b.eq #0x14                     x1: 0x00000000   0x00000001
ldr x9, [x2]                   x2: 0x80100000   0x80100000
ldr x9, [x3]                   x3: 0x80110000   0x80110000
ldr x9, [x4]                   x4: 0x80120000   0x80120000
b #0x48
ldr x9, [x2]
nop {14 times}
ldr x9, [x3]
ldr x9, [x4]
```

`Input 1` and `Input 2` are two states that exercise the two execution paths and have the same values for `x2`, `x3` and `x4`, hence the two states are observationally equivalent. Notice that all memory loads access cache set 0. Since the cache is 4-way associative and the cache is initially empty, we expect no eviction to occur.

Executions starting in `Input 2` behave as expected and terminate with the addresses of `x2`, `x3`, and `x4` in the final cache state. However, the execution from `Input 1` leads to a prevction, which causes the final cache state to only contain the addresses of `x3` and `x4`. The address of `x2` has been evicted even if the cache set is not full. Therefore the two states are distinguishable by the attacker. Our hypothesis is that the processor detects a short sequence of loads to the same cache set and anticipates more loads to the same cache set with no reuse of previously loaded values. It evicts the valid cache line in order to make space for more colliding lines. We note that these cache entries are not dirty and thus eviction is most likely a cheap operation. The execution of a `nop` sequence probably ensures that the first cache line fill is completed before the other addresses are accessed.

Offset-Dependent Behaviors. Our experiments identified further counterexamples that invalidate the observational model. In particular, the following counterexample also invalidates the observational model of Definition 5, where cache line offsets are not observable.

```
Program                          Input 1        Input 2

ldr x6, [x0]             x0 : 0x80108000     0x80108000
ldr x9, [x3, #4]         x3 : 0x800FFFFC     0x800FFFFC
ldr x2, [x16]            x16: 0x80100020     0x80100000
ldr x16,[x22]            x22: 0x8011FFF8     0x8011FFF8
ldr x9, [x22,#8]
```

This program consists of five consecutive load instructions. This program always produces five observations consisting of the cache tag and set index of the five addresses. `Input 1` and `Input 2` are observationally equivalent: they only differ for **`x16`**, which affects the address used for the third load, but the addresses `0x80100020` and `0x80100000` have the same cache tag and set index and only differ for the offset within the same cache line. However, these experiments lead to two distinguishable microarchitectural states. More specifically, execution from `Input 1` results in the filling of cache set 0, where the addresses of registers **`x0`**, **`x3`**, **`x16`** and **`x22`** `+ 8` are present in the cache, while executions from `Input 2` leads a cache state where the address of **`x0`** is not in the cache and has been probably evicted. This effect can be the result of the interaction between cache previction and cache bank collision [9,40], whose behavior depends on the cache offset. Notice that cache bank collision is undocumented for ARM Cortex-A53. Tromer et al. [46] have shown that such offset-dependent behaviors can make insecure side-channel countermeasures for AES that rely on making accesses to memory blocks (rather than addresses) key-independent.

7.4 Problems in Model Implementations

Additionally to microarchitectural features that invalidate the formal models, our experiments identified bugs of the implementation of the models: (1) the formalization of the ARMv8 instruction set used by the transpiler and (2) the module that inserts BIR observation statements into the transpiled binary to capture the observations that can be made according to a given observational model. Table 1.4 reports problems identified by the random program generator. Some of these failing experiments result in distinguishable states while others result in run-time exceptions. In fact, if the model predicts wrong memory accesses for a program then our framework can generate test inputs that cause accesses to unmapped memory regions. The example program in Fig. 4 exhibits both problems when executed with appropriate inputs.

Missing Observations. The second step of our framework translates binary programs to BIR and adds observations to reflect the observational model under validation. In order to generate observations that correspond to memory loads, we syntactically analyze the right-hand side of BIR assignments. For instance, for line $l2$ in Fig. 3 we generate an observation that depends on variable `X1` because the expression of assignment is `LOAD(MEM, X1)`. This approach is problematic when a memory load is immaterial for the result of an instruction. For example, `ldr xzr` and `ldr wzr` instructions load from memory to a register that is constantly zero. The following program loads from `x30` into `xzr`.

Program		Input 1	Input 2
`ldr xzr, [x30]`	`x30:`	`0x80000040`	`0x800000038`

The translation of this instruction is simply `[JMP next_addr]`: there is no assignment that loads from `x30` because the register `xzr` remains zero. Therefore, our model generates no observations and any two input states are observationally equivalent. The ARM specification does not clarify that the microarchitecture can skip the immaterial memory load. Our experiments show that this is not the case and therefore our implementation of the model is not correct. In fact, the program accesses cache set $\texttt{index}(0x80000040) = 1$ for `Input 1` and cache set $\texttt{index}(0x80000038) = 0$ for `Input 2`, which results in distinguishable states. Moreover, by not taking into account the memory access our framework generates some tests that set `x30` to unmapped addresses and cause run-time exceptions.

Flaw in HOL4 ARMv8 ISA Model. Our tool has identified a bug of the HOL4 ARMv8 ISA model. This model has been used in several projects [8,17] as the basis for formal analysis and is used by our framework to transform ARM programs to BIR programs. Despite its wide adoption, we identified a problem in the semantics of instructions *Compare and Branch on Zero* (CBZ) and *Compare and Branch on Non-Zero* (CBNZ). These instructions implement a conditional jump based on the comparison of the input register with zero. While CBZ jumps in case of equality, CBNZ jumps in case of inequality. However, our tests identified that CBNZ wrongly behaves as CBZ in the HOL4 model.

8 Related Work

Hardware Models. Verification approaches that take into account the underlying hardware architecture have to rely on a formal model of that architecture. Commercial instruction set architectures (ISAs) are usually specified mostly in natural language, and their formalization is an active research direction. For example, Goel et al. [24] formalize the ISA of x86 in ACL2, Morrisett et al. [37] model the x86 architecture in the Coq theorem prover, and Sarkar et al. [42] provide a formal semantics of the x86 multiprocessor ISA in HOL. Moreover, domain-specific languages for ISAs have been developed, such as the L3 language [19], which has been used to model the ARMv7 architecture. As another example, Siewiorek et al. [44] proposed the *Instruction-Set Processor* language for formalizing the semantics of the instructions of a processor.

Processor Verification and Validation. To gain confidence in the correctness of a processor model, it needs to be verified or validated against the actual hardware. This problem has received considerable attention lately. There are white-box approaches such as the formal verification that a processor model matches a hardware design [10,18]. These approaches differ from ours in that they try to give a formal guarantee that a processor model is a valid abstraction of the actual hardware, and to achieve that they require the hardware to be accessible as a white box. More similar to ours are black-box approaches that validate an abstract model by randomly generated instructions or based on dynamic instrumentation [20,29]. Combinations of formal verification and testing approaches for hardware verification and validation have also been considered [11].

In contrast to our work, all of the approaches above are limited to functional correctness, and validation is limited to single-instruction test cases, which we show to be insufficient for information flow properties. Going beyond these restrictions is the work of Campbell and Stark [12], who generate sequences of instructions as test cases, and go beyond functional correctness by including timing properties. Still, neither their models nor their approach is suitable to identify violations of information flow properties.

Validating Information Flow Properties. To the best of our knowledge, we present the first automated approach to validate processor models with respect to information flow properties. To this end, we build on the seminal works of McLean [35] on non-interference, Roscoe [41] on observational determinism, and Barthe et al. [7] on self-composition as a method for proving information flow properties. Most closely related is the work by Balliu et al. [6] on *relational analysis* based on *observational determinism*.

These approaches are based on the different observational models that have been proposed in the literature. For example, the program counter security model [36] has been used when the execution time depends on the control flow of the victim. Extensions of this model also make observable data that can affect execution time of an instruction, or memory addresses accessed by the program to model timing differences due to caching [4].

Many analysis tools use these observational models. Ct-verif [3] implements a sound information flow analysis by proving observational equivalence constructing a product program. CacheAudit [15] quantifies information leakage by using abstract interpretation.

The risks of using unsound models for such analyses have been demonstrated by the recent Spectre attack family [32], which exploits speculation to leak data through caches. Several other architectural details require special caution when using abstract models, as some properties assumed by the models could be unmet. For instance, cache clean operations do not always clean residual state in implementations of replacement policies [21]. Furthermore, many processors do not provide sufficient means to close all leakage, e.g., shared state cannot be cleaned properly on a context switch [22]. Finally, it has been shown that fixes relying on too specific assumptions can be circumvented by modifying the attack [43], and that attacks are possible even against formally verified software if the underlying processor model is unsound [28]. For these reasons, validation of formal models by directly measuring the hardware is of great importance.

9 Concluding Remarks

We presented Scam-V, a framework for automatic validation of observational models of side channels. Scam-V uses a novel combination of symbolic execution, relational analysis, and observational models to generate experiments. We evaluated Scam-V on the ARM Cortex-A53 processor and we invalidated all models of Sect. 2.3, i.e., those with observations that are cache-line-offset-independent.

Our results are summarized as follows: (i) in case of cache partitioning, the attacker can discover victim accesses to the other cache partitions due to the automatic data prefetcher; (ii) the Cortex-A53 prefetcher seems to respect 4K page boundaries, like in some Intel processors; (iii) a mechanism of Cortex-A53, which we called previction, can leak the time between accesses to the same cache set; (iv) the cache state is affected by the cache line offset of the accesses, probably due to undocumented cache bank collisions like in some AMD processors; (v) the formal ARMv8 model had a flaw in the implementation of CBNZ; (vi) our implementation of the observational model had a flaw in case of loads into the constant zero register. Moreover, since the microarchitectural features that lead to these findings are also available on other ARMv8 cores, including some that are affected by Spectre (e.g. Cortex A57), it is likely that similar behaviors can be observed on these cores, and that more powerful observational models, including those that take into account Spectre-like effects, may also be unsound.

These promising results show that Scam-V can support the identification of undocumented and security-relevant features of processors (like results (ii), (iii), and (iv)) and discover problems in the formal models (like results (v) and (vi)). In addition, users can drive test-case generation to conveniently explore classes of programs that they suspect would lead to side-channel leakage (like in result (i)). This process is enabled by path and term enumeration techniques as well as custom program generators. Moreover, Scam-V can aid vendors to validate implementations with respect to desired side-channel specifications.

Given the lack of vendor communication regarding security-relevant processor features, validation of abstract side-channel models is of critical importance. As a future direction of work, we are planning to extend Scam-V for other architectures (e.g. ARM Cortex-M0 based microcontrollers), noisy side channels (e.g. time and power consumption), and other side channels (e.g. cache replacement state). Moreover, we are investigating approaches to automatically repair an unsound observational model starting from the counterexamples, e.g., by adding state observations. Finally, the theory in Sect. 4 can be used to develop a certifying tool for verifying observational determinism.

Acknowledgement. We thank Matthias Stockmayer for his contributions to the symbolic execution engine in this work. This work has been supported by the TrustFull project financed by the Swedish Foundation for Strategic Research, the KTH CERCES

Center for Resilient Critical Infrastructures financed by the Swedish Civil Contingencies Agency, as well as the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

1. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 112–121. Springer, Heidelberg (2006). https://doi.org/10.1007/11935308_9
2. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 225–242. Springer, Heidelberg (2006). https://doi.org/10.1007/11967668_15
3. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security, pp. 53–70 (2016)
4. Almeida, J.B., Barbosa, M., Pinto, J.S., Vieira, B.: Formal verification of side-channel countermeasures using self-composition. Sci. Comput. Program. **78**(7), 796–812 (2013)
5. ARM Limited: Vulnerable ARM processors to Spectre attack. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. Accessed 2019
6. Balliu, M., Dam, M., Guanciale, R.: Automating information flow analysis of low level code. In: Proceedings of the Conference on Computer and Communications Security, CCS, pp. 1080–1091 (2014)
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
8. Baumann, C., Schwarz, O., Dam, M.: On the verification of system-level information flow properties for virtualized execution platforms. J. Cryptogr. Eng. **9**(3), 243–261 (2019). https://doi.org/10.1007/s13389-019-00216-4
9. Bernstein, D.J.: Cache-timing attacks on AES. Technical report (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
10. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together – formal verification of the VAMP. Int. J. Softw. Tools Technol. Transfer **8**(4–5), 411–430 (2006). https://doi.org/10.1007/s10009-006-0204-6
11. Bhadra, J., Abadir, M.S., Wang, L.C., Ray, S.: A survey of hybrid techniques for functional verification. Des. Test Comput. **24**(2), 112–122 (2007)
12. Campbell, B., Stark, I.: Randomised testing of a microprocessor model using SMT-solver state generation. Sci. Comput. Program. **118**, 60–76 (2016)
13. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. SIGPLAN Not. **35**(9), 268–279 (2000)
14. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: a tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. **18**(1), 4:1–4:32 (2015)
16. Fox, A.: L3: A Specification Language for Instruction Set Architectures. https://acjf3.github.io/l3. Accessed 2019

17. Fox, A., Myreen, M.O., Tan, Y.K., Kumar, R.: Verified compilation of CakeML to multiple machine-code targets. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP, pp. 125–137. Association for Computing Machinery, New York (2017)
18. Fox, A.: Formal specification and verification of ARM6. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 25–40. Springer, Heidelberg (2003). https://doi.org/10.1007/10930755_2
19. Fox, A.: Directions in ISA specification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 338–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_23
20. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
21. Ge, Q., Yarom, Y., Heiser, G.: Do hardware cache flushing operations actually meet our expectations. arXiv e-prints (2016)
22. Ge, Q., Yarom, Y., Li, F., Heiser, G.: Your processor leaks information-and there's nothing you can do about it. CoRR abs/1612.04474 (2017)
23. Godfrey, M.M., Zulkernine, M.: Preventing cache-based side-channel attacks in a cloud environment. IEEE Trans. Cloud Comput. **2**(4), 395–408 (2014)
24. Goel, S., Hunt Jr., W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: FMCAD, pp. 91–98. IEEE (2014)
25. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20. IEEE Computer Society (1982)
26. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: IEEE Symposium on Security and Privacy, pp. 75–87. IEEE Computer Society (1984)
27. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in JavaScript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 300–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_15
28. Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache storage channels: alias-driven attacks and verified countermeasures. In: IEEE Symposium on Security and Privacy, pp. 38–55. IEEE Computer Society (2016)
29. Hou, Z., Sanan, D., Tiu, A., Liu, Y., Hoa, K.C.: An executable formalisation of the SPARCv8 instruction set architecture: a case study for the LEON3 processor. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 388–405. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_24
30. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
31. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
32. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: S&P (2019)
33. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
34. Lindner, A., Guanciale, R., Metere, R.: TrABin: trustworthy analyses of binaries. Sci. Comput. Program. **174**, 72–89 (2019)

35. McLean, J.: Proving noninterference and functional correctness using traces. J. Comput. Secur. **1**(1), 37–58 (1992)
36. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: automatic detection and removal of control-flow side channel attacks. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 156–168. Springer, Heidelberg (2006). https://doi.org/10.1007/11734727_14
37. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: PLDI, pp. 395–404. ACM (2012)
38. Nemati, H., Buiras, P., Lindner, A., Guanciale, R., Jacobs, S.: Validation of abstract side-channel models for computer architectures (2020). https://arxiv.org/abs/2005.05254
39. Neve, M., Seifert, J.-P.: Advances on access-driven cache attacks on AES. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 147–162. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74462-7_11
40. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
41. Roscoe, A.W.: CSP and determinism in security modelling. In: IEEE Symposium on Security and Privacy, pp. 114–127. IEEE Computer Society (1995)
42. Sarkar, S., et al.: The semantics of x86-CC multiprocessor machine code. In: POPL, pp. 379–391. ACM (2009)
43. Schaik, S.V., Giuffrida, C., Bos, H., Razavi, K.: Malicious management unit: why stopping cache attacks in software is harder than you think. In: USENIX Security Symposium, pp. 937–954. USENIX Association (2018)
44. Siewiorek, D.P., Bell, G., Newell, A.C.: Computer Structures: Principles and Examples. McGraw-Hill Inc, New York (1982)
45. Taylor, G., Davies, P., Farmwald, M.: The TLB slice-a low-cost high-speed address translation mechanism. SIGARCH Comput. Archit. News **18**(2SI), 355–363 (1990)
46. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. J. Cryptol. **23**(1), 37–71 (2009). https://doi.org/10.1007/s00145-009-9049-y
47. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 62–76. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45238-6_6
48. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proceedings of the Conference on Computer and Communications Security, CCS, pp. 305–316. ACM (2012)