Alexandra Silva
K. Rustan M. Leino (Eds.)

LNCS 12759

Computer Aided Verification

33rd International Conference, CAV 2021
Virtual Event, July 20–23, 2021
Proceedings, Part I

Springer

OPEN ACCESS

Automated Safety Verification of Programs Invoking Neural Networks

Maria Christakis[1], Hasan Ferit Eniser[1]✉, Holger Hermanns[2,6], Jörg Hoffmann[2], Yugesh Kothari[1], Jianlin Li[3,2,7], Jorge A. Navas[4], and Valentin Wüstholz[5]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{maria,hferiser,ykothari}@mpi-sws.org
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
{hermanns,hoffmann}@cs.uni-saarland.de
[3] SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China
ljlin@ios.ac.cn
[4] SRI International, Menlo Park, USA
jorge.navas@sri.com
[5] ConsenSys, Kaiserslautern, Germany
valentin.wustholz@consensys.net
[6] Institute of Intelligent Software, Guangzhou, China
[7] University of Chinese Academy of Sciences, Beijing, China

Abstract. State-of-the-art program-analysis techniques are not yet able to effectively verify safety properties of heterogeneous systems, that is, systems with components implemented using diverse technologies. This shortcoming is pinpointed by programs invoking neural networks despite their acclaimed role as innovation drivers across many application areas. In this paper, we embark on the verification of system-level properties for systems characterized by interaction between programs and neural networks. Our technique provides a tight two-way integration of a program and a neural-network analysis and is formalized in a general framework based on abstract interpretation. We evaluate its effectiveness on 26 variants of a widely used, restricted autonomous-driving benchmark.

1 Introduction

Software is becoming increasingly *heterogeneous*. In other words, it consists of more and more diverse software components, implemented using different technologies such as neural networks, smart contracts, or web services. Here, we focus on programs invoking neural networks, in response to their prominent role in many upcoming application areas. Examples from the forefront of innovation include a controller of a self-driving car that interacts with a neural network identifying street signs [43,48], a banking system that consults a neural network for credit screening [3], or a health-insurance system that relies on a neural network to predict people's health needs [51]. There are growing concerns regarding the effects of integrating such heterogeneous technologies [40].

Despite these software advances, state-of-the-art program-analysis techniques cannot yet effectively reason across heterogeneous components. In fact, program

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12759, pp. 201–224, 2021.
https://doi.org/10.1007/978-3-030-81685-8_9

analyses today focus on homogeneous units of software in isolation; for instance, to check the robustness of a neural network (e.g., [37,27,36,66,65,64,57,41]), or safety of a program invoking a neural network while—conservatively, but imprecisely—treating the neural network as if it could return arbitrary values. This is a fundamental limitation of prevalent program-analysis techniques, and as a result, we cannot effectively analyze the interaction between diverse components of a heterogeneous system to check system properties.

Many properties of heterogeneous systems depend on components correctly interacting with each other. For instance, consider a program that controls the acceleration of an autonomous vehicle by invoking a neural network with the current direction, speed, and LiDAR image of the vehicle's surroundings. One might want to verify that the vehicle's speed never exceeds a given bound. Even such a seemingly simple property is challenging to verify automatically due to the mutual dependencies between the two components. On the one hand, the current vehicle direction and speed determine the feasible inputs to the neural network. On the other hand, the output of the neural network controls the vehicle acceleration, and thereby, the speed. To infer bounds on the speed (and ultimately prove the property), an automated analysis should therefore analyze how the two components interact.

Our approach. In this paper, we make the first step in verifying safety of heterogeneous systems, and more specifically, of *programs* invoking neural networks. Existing work on verification of neural networks has either focused on the network itself (e.g., with respect to robustness) or on *models* (e.g., expressed using differential equations) that invoke the network, for example as part of a hybrid system [24,59]. In contrast, our approach is designed for verifying safety of a C (or ultimately LLVM) program interacting with the network. In comparison to models, C programs are much more low-level and general, and therefore require an intricate combination of program and neural-network analyses.

More specifically, our approach proposes a symbiotic combination of a program and a neural-network analysis, both of which are based on abstract interpretation [18]. By treating the neural-network analysis as a specialized abstract domain of the program analyzer, we are able to use inferred invariants for the neural network to check system properties in the surrounding program. In other words, the program analysis becomes *aware* of the network's computation. For this reason, we also refer to the overall approach as *neuro-aware program analysis*. In fact, the program and neural-network analyses are co-dependent. The former infers sets of feasible inputs to the neural network, whereas the latter determines its possible outputs given the inferred inputs. Knowing the possible neural-network outputs, in turn, enables proving system safety.

We evaluate our approach on 26 variants of RACETRACK, a benchmark from related work that originates in AI autonomous decision making [4,5,32,46,52,53]. RACETRACK is about the problem of navigating a vehicle on a discrete map to a goal location without crashing into any obstacles. The vehicle acceleration (in discrete directions) is determined by a neural network, which is invoked by a controller responsible for actually moving the vehicle. In Sect. 4, we show the

effectiveness of our approach in verifying goal reachability and crash avoidance for 26 RACETRACK variants of varying complexity. These variants constitute a diverse set of verification tasks that differ both in the neural network itself and in how and for what purpose the program invokes the neural network.

Despite our evaluation being focused on this setting, the paper's contribution should not be mistaken as being about RACETRACK verification. Instead, it is about neuro-aware program analysis of heterogeneous systems for autonomous decision making. While RACETRACK is a substantially simplified blueprint for the autonomous-driving context, it features the crucial co-dependent program architecture that is characteristic across the entire domain.

Contributions. Overall, we make the following contributions:

1. We present the first *symbiotic* combination of program and neural-network analyses for verifying safety of heterogeneous systems.
2. We formalize neuro-aware program analysis in a general framework that uses specialized abstract domains.
3. We evaluate the effectiveness of our approach on 26 variants of a widely used, restricted autonomous-driving benchmark.

2 Overview

We now illustrate neuro-aware program analysis on a high level by describing the challenges in verifying safety for a variant of the RACETRACK benchmark. This variant serves as our running example for the class of programs that invoke one or more neural networks to perform a computation affecting program safety.

In general, RACETRACK is a heterogeneous system that simulates the problem of navigating a vehicle to a goal location on a discrete map without crashing into any obstacles. It consists of a neural network, which predicts the vehicle acceleration toward discrete directions, and a controller (implemented in C) that actually moves the vehicle on the map. Alg. 1 shows pseudo-code for our running example, a variant of RACETRACK that incorporates additional non-deterministic noise to make verification harder.

Line 1 non-deterministically selects a state from the map as the *currentState*, and line 2 assumes it is a start state for the vehicle, i.e., it is neither a goal nor an obstacle. On line 3, we initialize the *result* of navigating the vehicle as `stuck`, i.e., the vehicle neither crashes nor does it reach a goal. The loop on line 5 iterates until either a predefined number of steps N is reached or the vehicle is no longer `stuck` (i.e., crashed or at a goal state). The if-statement on line 6 adds non-determinism to the controller by either zeroing the vehicle acceleration or invoking the neural network (NN) to make a prediction. Such non-deterministic noise illustrates one type of variant we created to make the verification task more difficult (see Sect. 4.1 for more details on other variants used in our evaluation). Line 10 moves the vehicle to a new *currentState* according to *acceleration*, and the if-statement on line 11 determines whether the vehicle has crashed or reached a goal. The assertion on line 16 denotes the system properties of goal reachability

Algorithm 1: An example RACETRACK variant.

```
1  currentState ← ⋆
2  assume IsStartState(currentState)
3  result ← stuck
4  i ← 0
5  while i < N and result = stuck do
6    if ⋆ then
7      acceleration ← 0
8    else
9      acceleration ← NN(currentState)
10   currentState ← Move(currentState, acceleration)
11   if IsCrash(currentState) then
12     result ← crash
13   else if IsGoal(currentState) then
14     result ← goal
15   i ← i + 1
16 assert result = goal
```

and crash avoidance. In case this assertion does not hold but we do prove the *result* to be `stuck`, then we have only verified crash avoidance.

Note that these are *safety*, and not liveness, properties due to the bounded number of loop iterations (line 5)—N is 50 in our evaluation, thus making bounded model checking [8,15] intractable.

Challenges. Verifying safety of this heterogeneous system with state-of-the-art program-analysis techniques, such as abstract interpretation, is a challenging endeavor.

When considering the controller in isolation, the analysis is sound if it assumes that the neural network may return any output ($\top$). More specifically, the abstract interpreter can ignore the call to the neural network and simply havoc its return value (i.e., consider a non-deterministic value). In our running example, this means that any vehicle acceleration is possible from the perspective of the controller analysis. Therefore, it becomes infeasible to prove a system property such as crash avoidance. In fact, in Sect. 4, we show this to be the case even with the most precise controller analysis.

On the other hand, when considering the neural network in isolation, the analysis must assume that any input is possible ($\top$) even though this is not necessarily the case in the context of the controller. More importantly, without analyzing the controller, it becomes infeasible to prove properties about the entire system; as opposed to properties of the neural network, such as robustness.

Our approach. To address these issues, our approach tightly combines the controller and neural-network analyses in a two-way integration based on abstract interpretation.

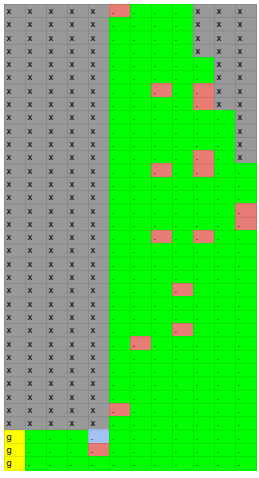
In general, an abstract interpreter infers invariants at each program state and verifies safety of an asserted property when it is implied by the invariant inferred in its pre-state. In the presence of loops, as in RACETRACK (line 5 in

Alg. 1), inference is performed for a number of iterations in order to reach a *fixpoint*, that is, infer invariants at each program state that do not change when performing additional loop iterations.

For our running example, to compute the fixpoint of the main loop, the controller analysis invokes the neural-network analysis instead of simply abstracting the call to the neural network by havocking its return value. The invariants inferred by the controller analysis in the pre-state of the call to the network are passed to the neural-network analysis; they are used to restrict the input space of the neural network. In turn, the invariants that are inferred by the neural-network analysis are returned to the controller analysis to restrict the output space. This exchange of verification results at analysis time significantly improves precision. By making the program analysis aware of the network's computation, neuro-aware program analysis is able to prove challenging safety properties of the entire system.

Our implementation combines off-the-shelf, state-of-the-art abstract interpreters, namely, CRAB [34] for the controller analysis and DEEPSYMBOL [41] or ERAN [27,56,57] for the neural-network analysis. CRAB[8] is a state-of-the-art analyzer for checking safety properties of LLVM bitcode programs. Its modular high-level architecture is similar to many other abstract interpreters, such as Astrée [9], Clousot [26], and Infer [11], and it supports a wide range of different abstract domains, such as Intervals [17], Polyhedra [19], and Boxes [33]. Specialized neural-network analyzers, such as DEEPSYMBOL or ERAN, have only very recently been developed to deal with the unique challenges of precisely checking robustness of neural networks; for instance, the challenge of handling the excessive number of "branches" induced by cascades of ReLU activations.

The technical details of this combination are presented in the following section. Note, however, that our technical framework does not prescribe a neural-network analysis that is necessarily based on abstract interpretation. Specifically, it could integrate any sound analysis that, given a set of (symbolic) input states, produces a set of output states over-approximating the return value of the neural network. We also discuss how our approach may integrate reasoning about other complex components, beyond neural networks. Our program analysis is also not inherently tied to CRAB, but could be performed by other abstract interpreters that use the same high-level architecture, such as Astrée [9].



The RACETRACK map on the right, which is borrowed from related work [5,32], shows the verification results achieved by our approach when combining CRAB and DEEPSYMBOL. Gray cells marked with 'x' denote obstacles, and yellow cells

[8] https://github.com/seahorn/crab

marked with 'g' denote goal locations. Recall from Alg. 1 that we can consider any cell, which is neither an obstacle nor a goal, as a possible start location.

In our evaluation, we run a separate analysis for each possible start state to identify all start locations from which the vehicle is guaranteed to reach a goal; in other words, the analysis tries to prove that *result* = `goal` holds (line 16 of Alg. 1) for each possible start location. Note that verifying a single start state already constitutes a challenging verification problem since, due to noise, the number of reachable states grows exponentially in the number of loop iterations (the vehicle can navigate to any feasible position). This setting of one start state is common in many reinforcement-learning environments, e.g., Atari games, Procgen [16], OpenAI Gym MiniGrid[9], etc.

Maps like the above are used throughout the paper to display the outcome of a verification process per cell. We color locations for which the process succeeds *green* in all shown maps. Similarly, we color states from which the vehicle might crash into an obstacle *red*; i.e., one or more states reachable from the start state may lead to a crash, and the analysis is not able to show that *result* $\neq$ `crash` holds before line 16. Finally, states from which the vehicle is guaranteed not to crash but might not reach a goal are colored in *blue*; i.e., the analysis is able to show that *result* $\neq$ `crash` holds before line 16, but it is not able to show that *result* $\neq$ `stuck` also holds.

As shown in the map, our approach is effective in verifying goal reachability and crash avoidance for the majority of start locations. Moreover, the verification results are almost identical when combining Crab with a different neural-network analyzer, namely Eran (see Sect. 4). Note that, since the analysis considers individual start states, the map may show a red start state that is surrounded by green start states. One explanation for this is that the vehicle never enters the red state from the surrounding green states or that it only enters the red state with a "safe" velocity and direction—imagine that the vehicle velocity when starting from the red state is always 2, whereas when entering it from green states, the velocity is always less. In general, whether a trajectory is safe largely depends on the neural-network behavior, which can be brittle.

3 Approach

As we discussed on a high level, our approach symbiotically combines an existing program analysis (PA) with a neural-network analysis (NNA). The result is a *neuro-aware program analysis* (NPA) that allows for precisely analyzing a program that invokes neural networks (see Fig. 1). In the following, we focus on a single network to keep the presentation simple. As shown in Fig. 1, the two existing analyses are extended to pass information both from PA to NNA (Φ in the diagram) and back (Ψ in the diagram).

In the following, we describe neuro-aware program analysis in more detail and elaborate on how the program analysis drives the neural-network analysis to verify safety properties of the containing heterogeneous system. Since the

[9] `https://github.com/maximecb/gym-minigrid`

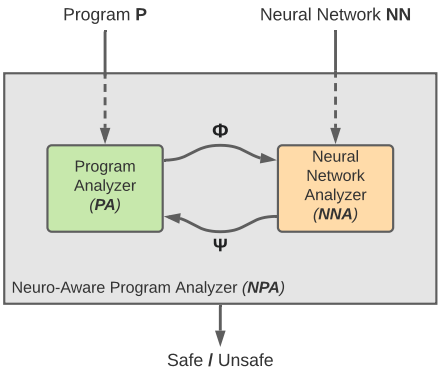


Figure 1: Overview of neuro-aware program analysis.

program analysis drives the neural-network analysis, we will explain the analysis in a top-down fashion by focusing on the program analysis before going into the details of the network analysis. In other words, our description of the program analysis assumes that we have a network analysis that over-approximates the behavior of the neural network.

3.1 Neuro-Aware Program Analysis

For our presentation, we assume imperative programs $\mathcal{P}$ with standard constructs, such as loops, function calls, arithmetic, and pointer operations (our implementation targets LLVM bitcode). In addition, we assume a special function call $\mathtt{o} := \mathtt{nn}(\mathtt{i}_1, \ldots, \mathtt{i}_n)$ that calls a neural network with input parameters $\mathtt{i}_1, \ldots, \mathtt{i}_n$ and returns the result of querying the network in return value $\mathtt{o}$. We also assume that the query does not have side effects on the program state. We denote programs $\mathcal{P}$ augmented with special calls to neural networks as $\mathcal{P}_{nn}$.

We assume an abstract domain D consisting of a set of abstract elements $d \in D$. Domain D is equipped with the usual binary operators $\langle \sqsubseteq, \sqcup, \sqcap, \bigtriangledown \bigtriangleup \rangle$, where the ordering between elements is given by $\sqsubseteq$. $\bot_D$ represents the smallest domain element and $\top_D$ the largest (smallest and largest relative to the ordering imposed by $\sqsubseteq$). The least upper bound (greatest lower bound) operator is denoted by $\sqcup$ ($\sqcap$). As usual, if the abstract domain is not finite or the number of elements is too large, then we also assume the domain to be equipped with widening ($\bigtriangledown$) and narrowing ($\bigtriangleup$) operators to ensure termination of the fixpoint computation. Moreover, we assume the abstract $\mathsf{forget} : D \times \overline{\mathcal{V}} \mapsto D$ operation that removes a set of variables from the abstract state, and its dual $\mathsf{project} : D \times \overline{\mathcal{V}} \mapsto D$ that projects the abstract state onto a set of variables. Finally, we assume the semantics function $[\![.]\!] : \mathcal{P} \mapsto D \mapsto D$ that, given a pre-state, computes the abstract semantics of a program to obtain its post-state; it does so recursively, by induction over the syntax of the program. We do not require that there exists a *Galois connection* [18] between the abstract domain D and the concrete domain

C. The only requirement is that D *over-approximates* C, i.e., $[\![.]\!]^C \subseteq \gamma \circ [\![.]\!] \circ \alpha$ where $[\![.]\!]^C$ is the concrete semantics and $\gamma : D \mapsto C$ and $\alpha : C \mapsto D$ are the concretization and abstraction functions, respectively.

We can then trivially define $\widehat{[\![.]\!]} : \mathcal{P}_{nn} \mapsto D \mapsto D$ to deal with $\mathcal{P}_{nn}$ as follows:

$$\widehat{[\![Cmd]\!]}(d) = \begin{cases} [\![\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})]\!](d) & \text{if } Cmd \equiv \mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n}) \\ [\![Cmd]\!](d) & \text{otherwise} \end{cases}$$

$$[\![\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})]\!](d) = \begin{cases} \bot_D & \text{if } d = \bot_D \\ \mathsf{forget}(d, o) & \text{otherwise} \end{cases}$$

However, this definition of $\widehat{[\![.]\!]}$ is not very useful since it conservatively approximates the neural network by havocking its return value $\mathtt{o}$.

To obtain a more precise approximation, we can integrate a designated neural-network analysis. Specifically, we view the neural-network analysis as another abstract domain D_{nn}, where, in practice, we do not require any other operation from D_{nn} except the transfer function for $\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})$ that soundly approximates the semantics of the neural network (see Sect. 3.2 for more details):

$$[\![\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})]\!](d) = \begin{cases} \bot_D & \text{if } d = \bot_D \\ \mathtt{let}\ d_{nn} = \mathsf{convert}(\mathsf{project}(d, i_1, \ldots, i_n))\ \mathtt{in} & \\ \mathtt{let}\ d'_{nn} = [\![\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})]\!]_{D_{nn}}(d_{nn})\ \mathtt{in} & \\ \mathsf{forget}(d, o) \sqcap \mathsf{convert}^{-1}(d'_{nn}) & \text{otherwise} \end{cases}$$

Intuitively, this more precise transfer function performs the following steps (unless d is $\bot_D$). First, it converts from D to D_{nn} to invoke the transfer function of D_{nn} on the converted value d_{nn}. It then havocs the return value o and conjoins the inferred return value after converting d'_{nn} back to D. In the above definition, functions $\mathsf{convert} : D \mapsto D_{nn}$ and $\mathsf{convert}^{-1} : D_{nn} \mapsto D$ convert from one abstract domain (D) to the other (D_{nn}) and back. We allow for conversions to result in loss of precision, that is, $\forall x \in D \cdot x \sqsubseteq \mathsf{convert}^{-1}(\mathsf{convert}(x))$.

It is important to realize here that the implementation of functions $\mathsf{convert}$ and $\mathsf{convert}^{-1}$, however precise, may still trigger a fatal loss of precision. After all, the abstract domains D and D_{nn} must also be expressive and precise enough to capture the converted values. For example, assume that, in a given program, the function call $\mathtt{o} := \mathtt{nn}(\mathtt{i_1}, \ldots, \mathtt{i_n})$ invokes a neural network to obtain the next move of a vehicle (encoded as a value from 0 to 7). Suppose the abstract return value d'_{nn} is the set of moves $\{1, 7\}$. In this case, given a domain D that cannot express these two moves as *disjuncts*, the implementation of function $\mathsf{convert}^{-1}$ has no choice but to abstract more coarsely; for instance, by expressing the return value as the interval $[1, \ldots, 7]$. Depending on the program, this may be too imprecise to prove safety. This happened to be the case for the RACETRACK variants we analyzed in Sect. 4, which is why we chose to use a disjunctive domain for the analysis; more specifically, we use Boxes [33], which allows us to track Boolean combinations of intervals.

Nevertheless, the considerations and approach described so far are still not precise enough for verifying the safety properties in the variants of RACETRACK that we consider. This is because the controller makes extensive use of (multi-dimensional) arrays, whose accesses are not handled precisely by the analysis. However, we observed that these arrays are initialized at the beginning of the system execution (for instance, to store maps indexed by x-y coordinates), and after initialization, they are no longer modified. Handling such array reads precisely is crucial in our context since over-approximation could conservatively indicate that the vehicle may crash into an obstacle.

Common array domains that summarize multiple elements using one or a small number of abstract elements fail to provide the needed precision. Even a fully expanded array domain [9] that separately tracks each array element loses precision if the index of an array read does not refer to a single array element; in such cases, the join of all overlapping elements will be returned. In addition, the Clang compiler—used to produce the LLVM bitcode that CRAB analyzes—desugars multi-dimensional arrays into single-dimensional arrays. This results in additional arithmetic operations (in particular, multiplications) for indexing the elements; these are also challenging to analyze precisely.

Interestingly, to address these challenges, we follow the same approach as for neural networks, in other words, by introducing a designated and very precise analysis to handle reads from these pre-initialized arrays. More formally, we introduce a new statement to capture such reads, $\mathtt{o} := \mathtt{ar}(\mathtt{i}_1, \ldots, \mathtt{i}_n)$, where i_k is the index for the k-th dimension of an n-dimensional array. Note that this avoids index conversions for multi-dimensional arrays since indices of each dimension are provided explicitly. Moreover, it is structurally very similar to the $\mathtt{nn}(\ldots)$ statement we introduced earlier. In particular, the specialized transfer function for D differs only in the two conversion functions and the specialized transfer function $[\![.]\!]_{D_{ar}}$:

$$[\![\mathtt{o} := \mathtt{ar}(\mathtt{i}_1, \ldots, \mathtt{i}_n)]\!](d) = \begin{cases} \bot_D & \text{if } d = \bot_D \\ \mathtt{let}\ d_{ar} = \mathsf{convert}_{\mathsf{ar}}(\mathsf{project}(d, i_1, \ldots, i_n))\ \mathtt{in} & \\ \mathtt{let}\ d'_{ar} = [\![\mathtt{o} := \mathtt{ar}(\mathtt{i}_1, \ldots, \mathtt{i}_n)]\!]_{D_{ar}}(d_{ar})\ \mathtt{in} & \\ \mathsf{forget}(d, o) \sqcap {\mathsf{convert}_{\mathsf{ar}}}^{-1}(d'_{ar}) & \text{otherwise} \end{cases}$$

To keep this transfer function simple, its input is a set of concrete indices and its output a set of concrete values that are retrieved by looking up the indexed elements in the array (after initialization). This makes it necessary for $\mathsf{convert}_{\mathsf{ar}}$ to concretize the abstract inputs to a disjunction of (concrete) tuples $(i_1, \ldots, i_n)$ for the read indices. Similarly, $\mathsf{convert}_{\mathsf{ar}}^{-1}$ converts the disjunction of (concrete) values back to an element of domain D.

Let us consider the concrete example in Fig. 2 to illustrate this more clearly. Line 1 initializes an array that is never again written to. On line 2, a non-deterministic value is assigned to variable `idx`, and the subsequent assume-statement constrains its value to be in the interval from 0 to 6. The assertion on line 5 checks that element `elem`, which is read from the array (on line 4),

```
1 int arr[] = {0, 1, 1, 2, 3, 5, 8, 13};
2 int idx = *;
3 assume(0 <= idx && idx <= 6);
4 int elem = arr[idx];
5 assert(elem < 13);
```

Figure 2: Example illustrating the specialized array domain.

is less than 13. Let us assume that we want to analyze the code by combining the numerical Intervals domain with our array domain D_{ar}; in other words, we assume D is instantiated with Intervals. In the pre-state of the array read, the analysis infers that the abstract value for `idx` is interval $[0, 6]$. When computing the post-state for the read operation, the analysis converts this interval to the concrete set of indices $\{0, 1, 2, 3, 4, 5, 6\}$ via $\mathsf{convert}_{\mathsf{ar}}$. The transfer function for the array domain then looks up the (concrete) elements for each index to obtain the (concrete) set $\{0, 1, 2, 3, 5, 8\}$. Before returning this set to the Intervals domain, the analysis applies $\mathsf{convert}_{\mathsf{ar}}^{-1}$ to obtain the abstract value $[0, 8]$. This post-state allows the numerical domain to prove the assertion.

Note that this array domain is not specific to controllers such as the one used in our RACETRACK variants. In fact, one could consider using it to more precisely analyze other programs with complex arrays that are initialized at runtime; a concrete example would be high-performance hash functions that often rely on fixed lookup tables.

Even more generally, the domains we sketched above suggest that our approach is also applicable to other scenarios; for instance, when a piece of code is too challenging to handle by a generic program analysis, and a simple summary or specification would result in unacceptable loss of precision.

3.2 Neural-Network Analysis

AI^2 [27] was the first tool and technique for verifying robustness of neural networks using abstract interpretation. ERAN is a successor of AI^2; it incorporates specialized transfer functions and abstract domains, such as DeepZ [56] (a variant of Zonotopes [28]) and DeepPoly [57] (a variant of Polyhedra [19]). Meanwhile, DEEPSYMBOL [41] extended AI^2 with a novel *symbolic-propagation* technique. In the following, we first provide an overview of the techniques in ERAN and DEEPSYMBOL. Then, we describe how their domains can be used to implement the specialized transfer function from D_{nn} that was introduced in Sect. 3.1. On a high level, even though we are not concerned with robustness properties in this work, we re-purpose components of these existing tools to effectively check safety properties of heterogeneous systems that use neural networks.

The main goal behind verifying robustness of a neural network is to provide guarantees about whether it is susceptible to adversarial attacks [31,12,50,44]. Such attacks slightly perturb an original input (e.g., an image) that is classified correctly by the network (e.g., as a dog) to obtain an adversarial input that is classified differently (e.g., as a cat). Given a concrete input (e.g., an image),

existing tools detect such local-robustness violations by expressing the set of all perturbed inputs (within a bounded distance from the original according to a metric, such as L_∞ [35]) and "executing" the neural network with this set of inputs to obtain a set of outcomes (or labels). The network is considered to be locally robust if there are no more than one possible outcome.

Existing techniques use abstract domains to express sets of inputs and outputs, and define specialized transfer functions to capture the operations (e.g., affine transforms and ReLUs) that are required for executing neural networks. For instance, ERAN uses the DeepPoly [57] domain that captures polyhedral constraints and incorporates custom transfer functions for affine transforms, ReLUs, and other common neural-network operations. DEEPSYMBOL propagates symbolic information on top of abstract domains [65,41] to improve its precision. The key insight is that neural networks make extensive use of operations that apply linear combinations of arguments, and symbolic propagation is able to track linear-equality relations between variables (e.g., activation values of neurons).

Both ERAN and DEEPSYMBOL have the following in common: they define an abstract semantics for reasoning about neural-network operations and for computing an abstract set of outcomes from a set of inputs. We leverage this semantics to implement the specialized transfer function $[\![\mathtt{o} := \mathtt{nn}(\mathtt{i}_1, ..., \mathtt{i}_n)]\!]_{D_{nn}}(d_{nn})$ from Sect. 3.1.

4 Experimental Evaluation

To evaluate our technique, we aim to answer the following research questions:

RQ1: How effective is our technique in verifying goal reachability and crash avoidance?
RQ2: How does the quality of the neural network affect the verification results?
RQ3: How does a more complex benchmark affect the verification results?
RQ4: How does the neural-network analyzer affect the verification results?

4.1 Benchmarks

We run our experiments on variants of RACETRACK, which is a popular benchmark in the AI community [4,5,32,46,52,53] and implements the pseudo-code from Alg. 1 in C (see Sect. 2 for a high-level overview of the benchmark).

The RACETRACK code[10] is significantly more complicated than the pseudo-code in Alg. 1 would suggest; more specifically, it consists of around 400 lines of C code and invokes a four-layer fully connected neural network—with 14 inputs, 9 outputs, and 64 neurons per hidden layer (using ReLU activation functions). To name a few sources of complexity, the *currentState* does not just denote a single value, but rather the position of the vehicle on the map, the magnitude and direction of its velocity, its distance to goal locations, and its distance to obstacles. As another example, the MOVE function runs the trajectory of the

[10] https://github.com/Practical-Formal-Methods/Racetrack-Benchmark

vehicle from the old to the new state while determining whether there are any obstacles in between.

For simplicity, the code does not use floating-point numbers to represent variables, such as position, velocity, acceleration, and distance. Therefore, the program analyzer does not need to reason about floating-point numbers, which is difficult for CRAB and most other tools. However, this does not semantically affect the neural network or its analysis, both of which do use floats. An interface layer converts the input features, tracked as integers in the controller, to normalized floats for the neural-network analysis. The output from the neural-network analysis is a set of floating-point intervals, which are logically mapped to integers representing discrete possible actions at a particular state.

We evaluate our approach on 26 variants of RACETRACK, which differ in the following aspects of the analyzed program or neural network.

Maps. We adopt three RACETRACK maps of varying complexity from related work [5,32], namely *barto-small* (BS) of size 12×35, *barto-big* (BB) of size 30×33, and *ring* (R) of size 45×50. The size of a map is measured in terms of its dimensions (i.e., width and height). The map affects not only the program behavior, but also the neural network that is invoked. The latter is due to the fact that we train custom networks for different maps.

Neural-network quality. The neural network (line 9 of Alg. 1) is trained, using reinforcement learning [60], to predict an acceleration given a vehicle state, that is, the position of the vehicle on the map, the magnitude and direction of its velocity, its distance to goal locations, and its distance to obstacles. As expected, the quality of the neural-network predictions depends on the amount of training. In our experiments, we use *well* (GOOD), *moderately* (MOD), and *poorly* (POOR) *trained* neural networks. We use the average reward at the end of the training process to control the quality. More details are provided in RQ2.

Noise. We complicate the RACETRACK benchmark by adding two sources of non-determinism, namely *environment* (ENV) and *neural-network* (NN) *noise*. Introducing such noise is common practice in reinforcement learning, for instance, when modeling real-world imperfections, like slippery ground.

When environment noise is enabled, the controller might zero the vehicle acceleration (in practice, with a small probability), instead of applying the acceleration predicted by the neural network for the current vehicle state. This source of non-determinism is implemented by the if-statement on line 6 of Alg. 1. Environment noise may be disabled for states that are too close to obstacles to allow the vehicle to avoid definite crashes by adjusting its course according to the neural-network predictions. The amount of environment noise is, therefore, controlled by the distance to an obstacle (OD) at which we disable it. For example, when $\text{OD} = 3$, environment noise is disabled for all vehicle states that are at most 3 cells away from any obstacle. Consequently, when $\text{OD} = 1$, we have a more noisy environment. Note that we do not consider $\text{OD} = 0$ since the environment would be too noisy to verify safety for any start state.

Note that environment noise is not meant to represent realistic noise, but rather to make the verification task more challenging. However, it is also not

entirely unrealistic and can be viewed as "necessarily rectifying steering course close to obstacles". Non-deterministically zeroing acceleration is inspired by related work [32].

For a given vehicle state, the neural network computes confidence values for each possible acceleration; these values sum up to 1. Normally, the predicted acceleration is the one with the largest confidence value, which however might not always be high. When neural-network noise is enabled, the network analyzer considers *any* acceleration for which the inferred upper bound on the confidence value is higher than a threshold ϵ. For example, when $\epsilon = 0.25$, any acceleration whose inferred confidence interval includes values greater than 0.25 might be predicted by the neural network. Consequently, for lower values of ϵ, the neural network becomes more noisy. Such probabilistic action selection is widely used in reinforcement learning [55].

Each of these two sources of noise—ENV and NN noise—renders the verification of a neural-network controller through enumeration of all possible execution paths intractable: due to the non-determinism, the number of execution paths from a given initial state grows exponentially with the number of control iterations (e.g., the main loop on line 5 of Alg. 1). In our RACETRACK experiments, the bound on the number of loop iterations is 50, and as a result, the number of execution paths from any given initial state quickly becomes very large. By statically reasoning about sets of execution paths, our approach is able to more effectively handle challenging verification tasks in comparison to exhaustive enumeration.

Lookahead functionality. We further complicate the benchmark by adding lookahead functionality (not shown in Alg. 1), which aims to counteract incorrect predictions of the neural network and prevent crashes. In particular, when this functionality is enabled, the controller simulates the vehicle trajectory when applying the acceleration predicted by the neural network a bounded number of additional times (denoted LA). For example, when LA = 3, the controller invokes the neural network 3 additional times to check whether the vehicle would crash if we were to consecutively apply the predicted accelerations. If this lookahead functionality indeed foresees a crash, then the controller reverses the direction of the acceleration that is predicted for the current vehicle state on line 9 of Alg. 1. Conceptually, the goal behind our lookahead functionality is similar to the one behind *shields* [2]. While lookahead is explicitly encoded in the program as code, shields provide a more declarative way for expressing such safeguards.

4.2 Implementation

For our implementation[11], we extended CRAB to support specialized abstract domains as described in Sect. 3. To integrate DEEPSYMBOL and ERAN, we implemented a thin wrapper around these tools to enable their analysis to start

[11] Our source code can be found at `https://github.com/Practical-Formal-Methods/clam-racetrack` and an installation at `https://hub.docker.com/r/practicalformalmethods/neuro-aware-verification`.

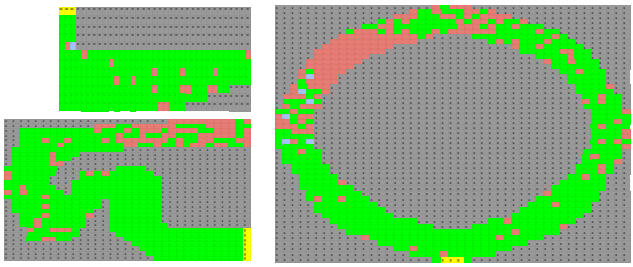


Figure 3: Verification results for RQ1, where ENV(OD = 3). The maps on the left are BS (top) and BB (bottom), and the map on the right is R.

from a set of abstract input states and return a set of abstract output states. Moreover, our wrappers provide control over the amount of neural-network noise (through threshold ϵ).

4.3 Setup

We use deep Q-learning [47] to train a neural network for each RACETRACK variant. We developed all training code in Python using the TensorFlow[12] and Torch[13] deep-learning libraries.

We configure CRAB to use the Boxes abstract domain [33], DEEPSYMBOL to use Intervals [18] with symbolic propagation [41], and ERAN to use DeepPoly [57]. When running the analysis, we did not specify a bound on the available time or memory; consequently, none of our analysis runs led to a time-out or mem-out. Regarding time, we report our results in the following, and regarding memory, our technique never exceeded 13.5GB when analyzing all start states of any map.

We performed all experiments on a 48-core Intel ® Xeon ® E7-8857 v2 CPU @ 3.6GHz machine with 1.5TB of memory, running Debian 10 (buster).

4.4 Results

We now present our experimental results for each research question.

RQ1: How effective is our technique in verifying goal reachability and crash avoidance? To evaluate the effectiveness of our technique in proving these system properties, we run it on the following benchmark variants: BS, BB, and R maps, GOOD neural networks, ENV noise with OD = 1, 2, 3, and LA = 0 (i.e., no lookahead). The verification results are shown in Figs. 3, 4, and 5 (see Sect. 2 for the semantics of cell colors). These results are achieved when combining CRAB with DEEPSYMBOL, but the combination with ERAN is comparable (see RQ4).

As shown in Fig. 3, for the vast majority of initial vehicle states, our technique is able to verify goal reachability and crash avoidance. This indicates that our integration of the controller and neural-network analyses is highly precise. As

[12] https://www.tensorflow.org
[13] http://torch.ch/

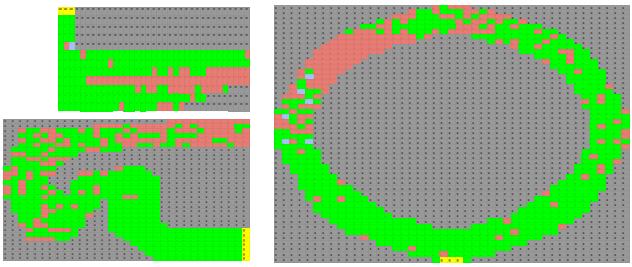


Figure 4: Verification results for RQ1, where ENV(OD = 2).

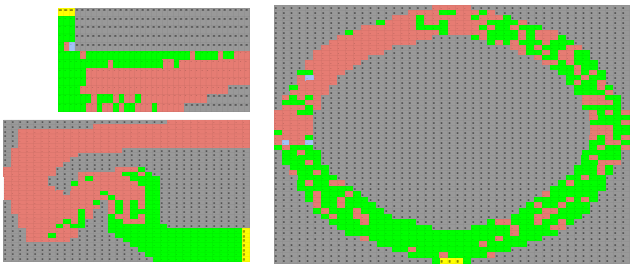


Figure 5: Verification results for RQ1, where ENV(OD = 1).

Table 1: Performance results for RQ1.

MAP	NN	NOISE	LA	NN ANALYZER	TOTAL TIME	AVG TIME	NN AVG TIME
BS	GOOD	ENV(OD = 3)	0	DEEPSYMBOL	1h20m34s	14m53s	30.2%
BB	GOOD	ENV(OD = 3)	0	DEEPSYMBOL	3h52m38s	18m55s	16.1%
R	GOOD	ENV(OD = 3)	0	DEEPSYMBOL	2h58m17s	11m33s	26.6%

expected, the more ENV noise we add (i.e., the smaller the OD values), the fewer states we prove safe (see Figs. 4 and 5).

Tab. 1 shows the performance of our technique. The first four columns of the table define the benchmark settings, the fifth the neural-network analyzer, and the last three show the total running time of our technique for all start states, the average time per state, and the percentage of this time spent on the neural-network analysis. Note that we measure the total time when running the verification tasks (for each start state) in parallel[14]; the average time per state is independent of any parallelization. We do not show performance results for different OD values since environment noise does not seem to have a significant impact on the analysis time.

Recall from Sect. 2 that, without our technique, it is currently only possible to verify properties of a heterogeneous system like RACETRACK by considering the controller in isolation, ignoring the call to the neural network, and havocking its return value. We perform this experiment for all of the above benchmark variants and find that CRAB alone is unable to prove goal reachability or crash

[14] https://doi.org/10.5281/zenodo.1146014

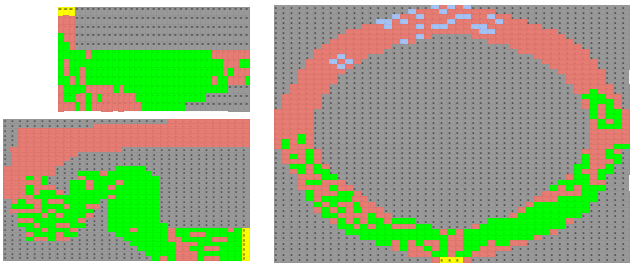


Figure 6: Verification results for RQ2, with MOD neural networks.

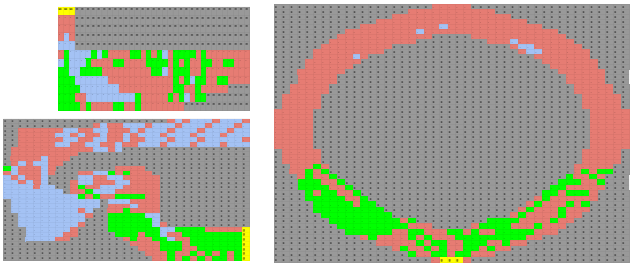


Figure 7: Verification results for RQ2, with POOR neural networks.

avoidance for *any* initial vehicle state; in other words, all states are *red*. This is the case even when replacing Boxes with Polyhedra—these two domains perform the most precise analyses in CRAB.

RQ2: How does the quality of the neural network affect the verification results? To evaluate this research question, we run our technique on the following benchmark variants: BS, BB, and R maps, MOD and POOR neural networks, ENV noise with OD = 3, and LA = 0. The verification results are shown in Figs. 6 and 7; they are achieved by combining CRAB with DEEPSYMBOL.

In deep Q-Learning (see Sect. 4.3), a neural network is trained by assigning positive or negative rewards to its predictions. A properly trained network learns to collect higher rewards over a run. Given this, we assess the quality of networks by considering average rewards over 100 runs from randomly selected starting states. If the network collects more than 70% of the maximum achievable reward, we consider it a GOOD agent. If it collects ca. 50% (or respectively, ca. 30%) of the maximum reward, we consider it a MOD (respectively, POOR) agent.

In comparison to Fig. 3, our technique proves safety of fewer states since the quality of the networks is worse. Analogously, more states are verified in Fig. 6 than in Fig. 7. Interestingly, for BB, our technique proves crash avoidance (blue cells) more often when using a POOR neural network (Fig. 7) instead of a MOD one (Fig. 6). We suspect that this is due to the randomness of the training process and the training policy, which penalizes crashes more than getting stuck; so, a POOR neural network might initially only try to avoid crashes.

Regarding performance, the analysis time fluctuates when using MOD and POOR neural networks. There is no pattern even when comparing the time across

Figure 8: Verification results for RQ3, where LA $= 0, 1, 3$ from left to right.

Table 2: Performance results for RQ3.

Map	NN	Noise	LA	NN Analyzer	Total Time	Avg Time	NN Avg Time
BS	MOD	ENV(OD = 3)	0	DeepSymbol	2h27m53s	27m35s	45.0%
BS	MOD	ENV(OD = 3)	1	DeepSymbol	8h04m40s	1h12m20s	14.9%
BS	MOD	ENV(OD = 3)	3	DeepSymbol	9h30m14s	1h47m05s	11.49%

different map sizes for equally trained networks. This is to be expected as neural networks may behave in unpredictable ways when not trained properly (e.g., the vehicle may drive in circles), which affects the performance of the analysis.

RQ3: How does a more complex benchmark affect the verification results? We complicate the benchmark by adding lookahead functionality, i.e., resulting in LA additional calls to the neural network per vehicle move (see Sect. 4.1 for more details). Since well trained neural networks would benefit less from this functionality, we use MOD networks in these experiments. In particular, we run our technique on the following benchmark variants: BS map, MOD neural networks, ENV noise with OD $= 3$, and LA $= 0, 1, 3$. The verification results are shown in Fig. 8; they are achieved by combining Crab with DeepSymbol.

As LA increases, the benchmark becomes more robust, yet more complex. We observe that, for larger values of LA, our technique retains its overall precision despite the higher complexity; e.g., there are states that are verified with LA $= 3$ or 1 but not with 0. However, there are also few states that are verified with LA $= 1$ but not with 3. In these cases, the higher complexity does have a negative impact on the precision of our analyses.

Tab. 2 shows the performance of our technique for these experiments. As expected, the analysis time increases as the benchmark complexity increases.

RQ4: How does the neural-network analyzer affect the verification results? We first compare DeepSymbol with Eran on the following benchmark variants: BS, BB, and R maps, GOOD neural networks, ENV noise with OD $= 3$, and LA $= 0$. The verification results achieved when combining Crab with Eran are shown in Fig. 9; compare this with Fig. 3 for DeepSymbol.

We observe the results to be comparable. With DeepSymbol, we color 216 cells green and 1 blue for BS, 455 green for BB, and 499 green and 6 blue for R. With Eran, the corresponding numbers are 214 cells green and 7 blue for BS, 459 green and 4 blue for BB, and 485 green and 71 blue for R. We observe similar results for other benchmark variants, but we omit them here.

Comparing the two neural-network analyzers becomes more interesting when we enable NN noise. More specifically, we run our technique on the following benchmark variants: BS, BB, and R maps, GOOD networks, NN noise with $\epsilon = 0.25$,

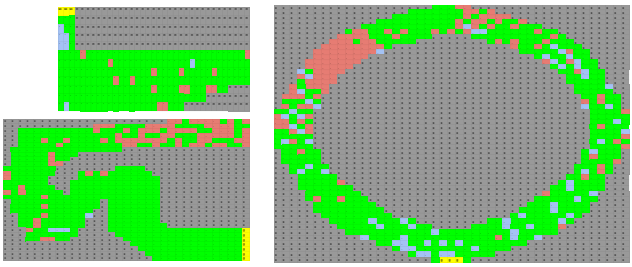


Figure 9: Verification results for RQ4 with ERAN, where ENV(OD = 3).

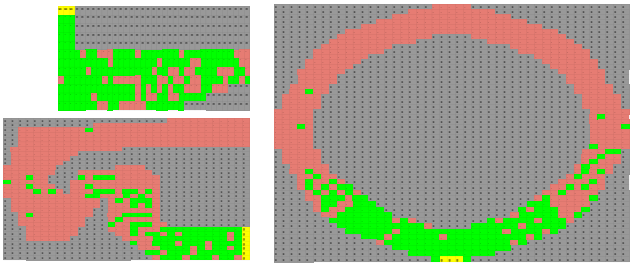


Figure 10: Verification results for RQ4 with DEEPSYMBOL, where NN($\epsilon = 0.25$).

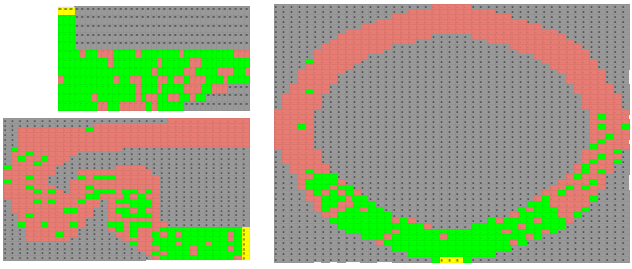


Figure 11: Verification results for RQ4 with ERAN, where NN($\epsilon = 0.25$).

and LA = 0. Fig. 10 shows the verification results when combining CRAB with DEEPSYMBOL, and Fig. 11 when combining CRAB with ERAN.

As shown in the figures, the verification results are slightly better with ERAN. In particular, with DEEPSYMBOL, we color 170 cells green for BS, 109 green for BB, and 195 green for R. With ERAN, the corresponding numbers are 181 cells green for BS, 131 green for BB, and 203 green for R. Despite this, the performance of our technique can differ significantly depending on whether we use DEEPSYMBOL or ERAN, as shown in Tab. 3. One could, consequently, imagine a setup where multiple neural-network analyzers are run in parallel for each verification task. If time is of the essence, we collect the results of the analyzer that terminates first. If it is more critical to prove safety, then we could combine the results of all analyzers once they terminate.

Table 3: Performance results for RQ4.

Map	NN	Noise	LA	NN Analyzer	Total Time	Avg Time	NN Avg Time
BS	GOOD	ENV(OD = 3)	0	DeepSymbol	1h20m34s	14m53s	30.2%
BS	GOOD	ENV(OD = 3)	0	Eran	43m21s	8m11s	38.4%
BB	GOOD	ENV(OD = 3)	0	DeepSymbol	3h52m38s	18m55s	16.1%
BB	GOOD	ENV(OD = 3)	0	Eran	3h26m17s	16m42s	57.2%
R	GOOD	ENV(OD = 3)	0	DeepSymbol	2h58m17s	11m33s	26.6%
R	GOOD	ENV(OD = 3)	0	Eran	4h38m03s	18m18s	53.8%
BS	GOOD	NN($\epsilon = 0.25$)	0	DeepSymbol	1h26m09s	15m41s	36.7%
BS	GOOD	NN($\epsilon = 0.25$)	0	Eran	45m37s	8m24s	45.0%
BB	GOOD	NN($\epsilon = 0.25$)	0	DeepSymbol	2h52m50s	13m24s	20.7%
BB	GOOD	NN($\epsilon = 0.25$)	0	Eran	2h59m48s	14m10s	64.7%
R	GOOD	NN($\epsilon = 0.25$)	0	DeepSymbol	2h01m18s	7m57s	26.4%
R	GOOD	NN($\epsilon = 0.25$)	0	Eran	3h21m32s	13m11s	54.3%

5 Related Work

The program-analysis literature provides countless examples of powerful analysis combinations. To name a few, dynamic symbolic execution [29,10] and hybrid fuzzing [45,58,69] combine random testing and symbolic execution, numerous tools broadly combine static and dynamic analysis [6,20,21,49,30,13,14,22,61], and many tools combine different types of static analysis [7,1,34]. In contrast to neuro-aware program analysis, almost all these tools target homogeneous, instead of heterogeneous, systems. Concerto [61] is a notable exception that targets applications using frameworks such as Spring and Struts. It combines abstract and concrete interpretation, where, on a high level, concrete interpretation is used to analyze framework code, whereas abstract interpretation is used for application code. Instead of building on existing analyzers, as in our work, Concerto introduces a designated technique for analyzing framework code.

There is recent work that focuses specifically on verifying hybrid systems with DNN controllers [24,59]. Unlike in our work, they do not analyze programs that interact with the network, but models; in one case, ordinary differential equations describing the hybrid system [24], and in the other, a mathematical model of a LiDAR image processor [59]. In this context of hybrid systems with DNN controllers, there is also work that takes a falsification approach to the same problem [62,70,23]. They generate corner test cases that cause the system to violate a system-level specification. Moreover, existing reachability analyses for neural networks [25,42,67,68] consider linear or piecewise-linear systems, instead of programs invoking them.

Kazak et al. [39] recently proposed Verily, a technique for verifying systems based on deep reinforcement learning. Such systems have been used in various contexts, such as adaptive video streaming, cloud resource management, and Internet congestion control. Verily builds on Marabou [38], a verification tool for neural networks, and aims to ensure that a system achieves desired service-level objectives (expressed as safety or liveness properties). Other techniques use abstract interpretation to verify robustness [27,57,41] or fairness properties [63] of neural networks. Furthermore, there are several existing techniques for check-

ing properties of neural networks using SMT solvers [37,38,36] and global optimization techniques [54]. In contrast to our approach, they focus on verifying properties of the network in isolation, i.e., without considering a program that queries it. However, we re-purpose two of the above analyzers [57,41] to infer invariants over the neural-network outputs. Gros et al. [32] make use of statistical model checking to obtain quality-assurance reports for a neural network in a noisy environment. Their approach provides probabilistic guarantees about checked properties, instead of definite ones like in our work, and also does not analyze a surrounding system.

6 Conclusion

Many existing software systems are already heterogeneous, and we expect the number of such systems to grow further. In this paper, we present a novel approach to verifying safety properties of such systems that symbiotically combines existing program and neural-network analyzers. Neuro-aware program analysis is able to effectively prove non-trivial system properties of programs invoking neural networks, such as the 26 variants of RACETRACK.

Acknowledgements. We are grateful to the reviewers for their constructive feedback. This work has been supported by DFG Grant 389792660 as part of TRR 248 (see `https://perspicuous-computing.science`). Jorge Navas has been supported by NSF Grant 1816936. Holger Hermanns and Jianlin Li have been supported by Guangdong Province Science Grant 2018B010107004.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: CAV. LNCS, vol. 7358, pp. 672–678. Springer (2012)
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. pp. 2669–2678. AAAI (2018)
3. Babaev, D., Savchenko, M., Tuzhilin, A., Umerenkov, D.: E.T.-RNN: Applying deep learning to credit loan applications. In: KDD. pp. 2183—2190. ACM (2019)
4. Baier, C., Christakis, M., Gros, T.P., Groß, D., Gumhold, S., Hermanns, H., Hoffmann, J., Klauck, M.: Lab conditions for research on explainable automated decisions. In: TAILOR. LNCS, vol. 12641, pp. 83–90. Springer (2020)
5. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artif. Intell. **72**, 81–138 (1995)
6. Beyer, D., Chlipala, A.J., Majumdar, R.: Generating tests from counterexamples. In: ICSE. pp. 326–335. IEEE Computer Society (2004)
7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: FSE. pp. 57–67. ACM (2012)
8. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)

9. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI. pp. 196–207. ACM (2003)
10. Cadar, C., Engler, D.R.: Execution generated test cases: How to make systems code crash itself. In: SPIN. LNCS, vol. 3639, pp. 2–23. Springer (2005)
11. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NFM. LNCS, vol. 9058, pp. 3–11. Springer (2015)
12. Carlini, N., Wagner, D.A.: Defensive distillation is not robust to adversarial examples. CoRR **abs/1607.04311** (2016)
13. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: FM. LNCS, vol. 7436, pp. 132–146. Springer (2012)
14. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: ICSE. pp. 144–155. ACM (2016)
15. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. FMSD **19**, 7–34 (2001)
16. Cobbe, K., Hesse, C., Hilton, J., Schulman, J.: Leveraging procedural generation to benchmark reinforcement learning. In: ICML. PMLR, vol. 119, pp. 2048–2056. PMLR (2020)
17. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: ISOP. pp. 106–130. Dunod (1976)
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96. ACM (1978)
20. Csallner, C., Smaragdakis, Y.: Check 'n' Crash: Combining static checking and testing. In: ICSE. pp. 422–431. ACM (2005)
21. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. TOSEM **17**, 1–37 (2008)
22. Czech, M., Jakobs, M.C., Wehrheim, H.: Just test what you cannot verify! In: FASE. LNCS, vol. 9033, pp. 100–114. Springer (2015)
23. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: NFM. LNCS, vol. 10227, pp. 357–372. Springer (2017)
24. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: HSCC. pp. 157–168. ACM (2019)
25. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Learning and verification of feedback control systems using feedforward neural networks. In: ADHS. IFAC-PapersOnLine, vol. 51, pp. 151–156. Elsevier (2018)
26. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: FoVeOOS. LNCS, vol. 6528, pp. 10–30. Springer (2010)
27. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: Safety and robustness certification of neural networks with abstract interpretation. In: S&P. pp. 3–18. IEEE Computer Society (2018)
28. Ghorbal, K., Goubault, E., Putot, S.: The Zonotope abstract domain Taylor1+. In: CAV. LNCS, vol. 5643, pp. 627–633. Springer (2009)
29. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI. pp. 213–223. ACM (2005)

30. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL. pp. 43–56. ACM (2010)
31. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: ICLR (2015)
32. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Deep statistical model checking. In: FORTE. LNCS, vol. 12136, p. 12136. Springer (2020)
33. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: SAS. LNCS, vol. 6337, pp. 287–303. Springer (2010)
34. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
35. Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press (2012)
36. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV. LNCS, vol. 10426, pp. 3–29. Springer (2017)
37. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV. LNCS, vol. 10426, pp. 97–117. Springer (2017)
38. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The Marabou framework for verification and analysis of deep neural networks. In: CAV. LNCS, vol. 11561, pp. 443–452. Springer (2019)
39. Kazak, Y., Barrett, C.W., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: NetAI@SIGCOMM. pp. 83–89. ACM (2019)
40. Larus, J., Hankin, C., Carson, S.G., Christen, M., Crafa, S., Grau, O., Kirchner, C., Knowles, B., McGettrick, A., Tamburri, D.A., Werthner, H.: When computers decide: European recommendations on machine-learned automated decision making. Tech. rep. (2018)
41. Li, J., Liu, J., Yang, P., Chen, L., Huang, X., Zhang, L.: Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In: SAS. LNCS, vol. 11822, pp. 296–319. Springer (2019)
42. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. CoRR **abs/1706.07351** (2017)
43. Luo, H., Yang, Y., Tong, B., Wu, F., Fan, B.: Traffic sign recognition using a multi-task convolutional neural network. Trans. Intell. Transp. Syst. **19**, 1100–1111 (2018)
44. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: ICLR. OpenReview.net (2018)
45. Majumdar, R., Sen, K.: Hybrid concolic testing. In: ICSE. pp. 416–426. IEEE Computer Society (2007)
46. McMahan, H.B., Gordon, G.J.: Fast exact planning in Markov decision processes. In: ICAPS. pp. 151–160. AAAI (2005)
47. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing Atari with deep reinforcement learning. CoRR **abs/1312.5602** (2013)
48. Nassi, B., Mirsky, Y., Nassi, D., Ben-Netanel, R., , Drokin, O., Elovici, Y.: Phantom of the ADAS: Securing advanced driver-assistance systems from split-second phantom attacks. In: CCS. pp. 293–308. ACM (2020)
49. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI project: Software property checking via static analysis and testing. In: TACAS. LNCS, vol. 5505, pp. 178–181. Springer (2009)

50. Papernot, N., McDaniel, P.D., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: EuroS&P. pp. 372–387. IEEE Computer Society (2016)
51. Pham, T., Tran, T., Phung, D.Q., Venkatesh, S.: Predicting healthcare trajectories from medical records: A deep learning approach. J. Biomed. Informatics **69**, 218–229 (2017)
52. Pineda, L.E., Lu, Y., Zilberstein, S., Goldman, C.V.: Fault-tolerant planning under uncertainty. In: IJCAI. pp. 2350–2356. IJCAI/AAAI (2013)
53. Pineda, L.E., Zilberstein, S.: Planning under uncertainty using reduced models: Revisiting determinization. In: ICAPS. pp. 217–225. AAAI (2014)
54. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: IJCAI. pp. 2651–2659. ijcai.org (2018)
55. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR **abs/1707.06347** (2017)
56. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: NeurIPS. pp. 10825–10836 (2018)
57. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**, 41:1–41:30 (2019)
58. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. The Internet Society (2016)
59. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: HSCC. pp. 147–156. ACM (2019)
60. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (2018)
61. Toman, J., Grossman, D.: CONCERTO: A framework for combined concrete and abstract interpretation. PACMPL **3**(POPL), 43:1–43:29 (2019)
62. Tuncali, C.E., Fainekos, G., Ito, H., Kapinski, J.: Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In: IV. pp. 1555–1562. IEEE Computer Society (2018)
63. Urban, C., Christakis, M., Wüstholz, V., Zhang, F.: Perfectly parallel fairness certification of neural networks. PACMPL **4**, 185:1–185:30 (2020)
64. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: NeurIPS. pp. 6369–6379 (2018)
65. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Security. pp. 1599–1614. USENIX (2018)
66. Wicker, M., Huang, X., Kwiatkowska, M.: Feature-guided black-box safety testing of deep neural networks. In: TACAS. LNCS, vol. 10805, pp. 408–426. Springer (2018)
67. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. TNNLS **29**, 5777–5783 (2018)
68. Xiang, W., Tran, H., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In: ACC. pp. 1574–1579. IEEE Computer Society (2018)
69. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Security. pp. 745–761. USENIX (2018)
70. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte carlo tree search. Trans. Comput. Aided Des. Integr. Circuits Syst. **37**, 2894–2905 (2018)