An Inductive Synthesis Framework for Verifiable Reinforcement Learning

He Zhu
Galois, Inc.
USA
hezhu@galois.com

Zikang Xiong
Purdue University
USA
xiong84@purdue.edu

Stephen Magill
Galois, Inc.
USA
stephen@galois.com

Suresh Jagannathan
Purdue University
USA
suresh@cs.purdue.edu

Abstract
Despite the tremendous advances that have been made in the last decade on developing useful machine-learning applications, their wider adoption has been hindered by the lack of strong assurance guarantees that can be made about their behavior. In this paper, we consider how formal verification techniques developed for traditional software systems can be repurposed for verification of reinforcement learning-enabled ones, a particularly important class of machine learning systems. Rather than enforcing safety by examining and altering the structure of a complex neural network implementation, our technique uses blackbox methods to synthesizes deterministic programs, simpler, more interpretable, approximations of the network that can nonetheless guarantee desired safety properties are preserved, even when the network is deployed in unanticipated or previously unobserved environments. Our methodology frames the problem of neural network verification in terms of a counterexample and syntax-guided inductive synthesis procedure over these programs. The synthesis procedure searches for both a deterministic program and an inductive invariant over an infinite state transition system that represents a specification of an application’s control logic. Additional specifications defining environment-based constraints can also be provided to further refine the search space. Synthesized programs deployed in conjunction with a neural network implementation dynamically enforce safety conditions by monitoring and preventing potentially unsafe actions proposed by neural policies. Experimental results over a wide range of cyber-physical applications demonstrate that software-inspired formal verification techniques can be used to realize trustworthy reinforcement learning systems with low overhead.

CCS Concepts • Software and its engineering → Automatic programming; Formal software verification.

Keywords Program Verification, Program Synthesis, Invariant Inference, Reinforcement Learning, Runtime Shielding

1 Introduction
Neural networks have proven to be a promising software architecture for expressing a variety of machine learning applications. However, non-linearity and stochasticity inherent in their design greatly complicate reasoning about their behavior. Many existing approaches to verifying [17, 23, 31, 37] and testing [43, 45, 48] these systems typically attempt to tackle implementations head-on, reasoning directly over the structure of activation functions, hidden layers, weights, biases, and other kinds of low-level artifacts that are far-removed from the specifications they are intended to satisfy. Moreover, the notion of safety verification that is typically considered in these efforts ignore effects induced by the actual environment in which the network is deployed, significantly weakening the utility of any safety claims that are actually proven. Consequently, effective verification methodologies in this important domain still remains an open problem.

To overcome these difficulties, we define a new verification toolchain that reasons about correctness extensionally, using a syntax-guided synthesis framework [5] that generates a simpler and more malleable deterministic program guaranteed to represent a safe control policy of a reinforcement
learning (RL)-based neural network, an important class of machine learning systems, commonly used to govern cyber-physical systems such as autonomous vehicles, where high assurance is particularly important. Our synthesis procedure is designed with verification in mind, and is thus structured to incorporate formal safety constraints drawn from a logical specification of the control system the network purports to implement, along with additional salient environment properties relevant to the deployment context. Our synthesis procedure treats the neural network as an oracle, extracting a deterministic program \( \mathcal{P} \) intended to approximate the policy actions implemented by the network. Moreover, our procedure ensures that a synthesized program \( \mathcal{P} \) is formally verified safe. To this end, we realize our synthesis procedure via a counterexample guided inductive synthesis (CEGIS) loop [5] that eliminates any counterexamples to safety of \( \mathcal{P} \). More importantly, rather than repairing the network directly to satisfy the constraints governing \( \mathcal{P} \), we instead treat \( \mathcal{P} \) as a safety shield that operates in tandem with the network, overriding network-proposed actions whenever such actions can be shown to lead to a potentially unsafe state. Our shielding mechanism thus retains performance, provided by the neural policy, while maintaining safety, provided by the program. Our approach naturally generalizes to infinite state systems with a continuous underlying action space. Taken together, these properties enable safety enforcement of RL-based neural networks without having to suffer a loss in performance to achieve high assurance. We show that over a range of cyber-physical applications defining various kinds of control systems, the overhead of runtime assurance is nominal, less than a few percent, compared to running an unproven, and thus potentially unsafe, network with no shield support. This paper makes the following contributions:

1. We present a verification toolchain for ensuring that the control policies learned by an RL-based neural network are safe. Our notion of safety is defined in terms of a specification of an infinite state transition system that captures, for example, the system dynamics of a cyber-physical controller.

2. We develop a counterexample-guided inductive synthesis framework that treats the neural control policy as an oracle to guide the search for a simpler deterministic program that approximates the behavior of the network but which is more amenable for verification. The synthesis procedure differs from prior efforts [9, 47] because the search procedure is bounded by safety constraints defined by the specification (i.e., state transition system) as well as a characterization of specific environment conditions defining the application’s deployment context.

3. We use a verification procedure that guarantees actions proposed by the synthesized program always lead to a state consistent with an inductive invariant of the original specification and deployed environment context. This invariant defines an inductive property that separates all reachable (safe) and unreachable (unsafe) states expressible in the transition system.

4. We develop a runtime monitoring framework that treats the synthesized program as a safety shield [4], overriding proposed actions of the network whenever such actions can cause the system to enter into an unsafe region.

We present a detailed experimental study over a wide range of cyber-physical control systems that justify the utility of our approach. These results indicate that the cost of ensuring verification is low, typically on the order of a few percent. The remainder of the paper is structured as follows. In the next section, we present a detailed overview of our approach. Sec. 3 formalizes the problem and the context. Details about the synthesis and verification procedure are given in Sec. 4. A detailed evaluation study is provided in Sec. 5. Related work and conclusions are given in Secs. 6 and 7, resp.

2 Motivation and Overview

To motivate the problem and to provide an overview of our approach, consider the definition of a learning-enabled controller that operates an inverted pendulum. While the specification of this system is simple, it is nonetheless representative of a number of practical control systems, such as Segway transporters and autonomous drones, that have thus far proven difficult to verify, but for which high assurance is very desirable.

2.1 State Transition System

We model an inverted pendulum system as an infinite state transition system with continuous actions in Fig. 1. Here, the pendulum has mass \( m \) and length \( l \). A system state is \( s = [\eta, \omega]^T \) where \( \eta \) is the pendulum’s angle and \( \omega \) is its angular velocity. A controller can use a 1-dimensional continuous control action \( a \) to maintain the pendulum upright.

![Figure 1. Inverted Pendulum State Transition System. The pendulum has mass \( m \) and length \( l \). A system state is \( s = [\eta, \omega]^T \) where \( \eta \) is the angle and \( \omega \) is its angular velocity. A continuous control action \( a \) maintains the pendulum upright.](image)
system dynamics, the change of rate of \( s \), denoted as \( \dot{s} \), to transition every \( t \) time period, with the conjecture that the control action \( a \) is a constant during each discrete time interval. Using Euler’s method, for example, a transition from state \( s_k = s(kt) \) at time \( kt \) to time \( kt + t \) is approximated as \( s(kt + t) = s(kt) + \dot{s}(kt) \times t \). We specify the change of rate \( \dot{s} \) using the differential equation shown in Fig. 1.\(^1\) Intuitively, the control action \( a \) is allowed to affect the change of rate of \( \eta \) and \( \omega \) to balance a pendulum. Thus, small values of \( a \) result in small swing and velocity changes of the pendulum, actions that are useful when the pendulum is upright (or nearly so), while large values of \( a \) contribute to larger changes in swing and velocity, actions that are necessary when the pendulum enters a state where it may risk losing balance. In case the system dynamics are unknown, we can use known algorithms to infer dynamics from online experiments [1].

Assume the state transition system of the inverted pendulum starts from a set of initial states \( S_0 \):

\[
S_0 : \{ (\eta, \omega) \mid -\theta \leq \eta \leq \theta \land -\dot{\theta} \leq \dot{\omega} \leq \dot{\theta} \}
\]

The global safety property we wish to preserve is that the pendulum never falls down. We define a set of unsafe states \( S_u \) of the transition system (colored in yellow in Fig. 1):

\[
S_u : \{ (\eta, \omega) \mid -\theta < \eta < \theta \land -\dot{\theta} < \dot{\omega} < \dot{\theta} \}
\]

We assume the existence of a neural network control policy \( \pi_w \) \( : \mathbb{R}^2 \rightarrow \mathbb{R} \) that executes actions over the pendulum, whose weight values of \( w \) are learned from training episodes. This policy is a state-dependent function, mapping a 2-dimensional state \( s \) (\( \eta \) and \( \omega \)) to a control action \( a \). At each transition, the policy mitigates uncertainty through feedback over state variables in \( s \).

**Environment Context.** An environment context \( C[-] \) defines the behavior of the application, where \([-] \) is left open to deploy a reasonable neural controller \( \pi_w \). The actions of the controller are dictated by constraints imposed by the environment. In its simplest form, the environment is simply a state transition system. In the pendulum example, this would be the equations given in Fig. 1, parameterized by pendulum mass and length. In general, however, the environment may include additional constraints (e.g., a constraining bounding box that restricts the motion of the pendulum beyond the specification given by the transition system in Fig. 1).

### 2.2 Synthesis, Verification and Shielding

In this paper, we assume that a neural network is trained using a state-of-the-art reinforcement learning strategy [28, 40]. Even though the resulting neural policy may appear to work well in practice, the complexity of its implementation makes it difficult to assert any strong and provable claims about its correctness since the neurons, layers, weights and biases are far-removed from the intent of the actual controller. We found that state-of-the-art neural network verifiers [17, 23] are ineffective for verification of a neural controller over an infinite time horizon with complex system dynamics.

**Framework.** We construct a policy interpretation mechanism to enable verification, inspired by prior work on imitation learning [36, 38] and interpretable machine learning [9, 47]. Fig. 2 depicts the high-level framework of our approach. Our idea is to synthesize a deterministic policy program from a neural policy \( \pi_w \), approximating \( \pi_w \) (which we call an oracle) with a simpler structural program \( P \). Like \( \pi_w \), \( P \) takes as input a system state and generates a control action \( a \). To this end, \( P \) is simulated in the environment used to train the neural policy \( \pi_w \), to collect feasible states. Guided by \( \pi_w \)’s actions on such collected states, \( P \) is further improved to resemble \( \pi_w \).

The goal of the synthesis procedure is to search for a deterministic program \( P^* \) satisfying both (1) a quantitative specification such that it bears reasonably close resemblance to its oracle so that allowing it to serve as a potential substitute is a sensible notion, and (2) a desired logical safety property such that when operation the unsafe states defined in the environment \( C \) cannot be reached. Formally,

\[
P^* = \underset{P \models \text{Safe}(C; [\mathcal{H}])}{\arg \max} d(\pi_w, P, C)
\]

where \( d(\pi_w, P, C) \) measures proximity of \( P \) with its neural oracle in an environment \( C ; [\mathcal{H}] \) defines a search space for \( P \) with prior knowledge on the shape of target deterministic programs; and, \( \text{Safe}(C; [\mathcal{H}]) \) restricts the solution space to a set of safe programs. A program \( P \) is safe if the safety of the transition system \( C[P] \), the deployment of \( P \) in the environment \( C \), can be formally verified.

The novelty of our approach against prior work on neural policy interpretation [9, 47] is thus two-fold:

1. We bake in the concept of safety and formal safety verification into the synthesis of a deterministic program from a neural policy as depicted in Fig. 2. If a candidate program is not safe, we rely on a counterexample-guided inductive synthesis loop to improve our synthesis outcome to enforce the safety conditions imposed by the environment.
2. We allow \( P \) to operate in tandem with the high-performing neural policy. \( P \) can be viewed as capturing an inductive
invariant of the state transition system, which can be used as a shield to describe a boundary of safe states within which the neural policy is free to make optimal control decisions. If the system is about to be driven out of this safety boundary, the synthesized program is used to take an action that is guaranteed to stay within the space subsumed by the invariant. By allowing the synthesis procedure to treat the neural policy as an oracle, we constrain the search space of feasible programs to be those whose actions reside within a proximate neighborhood of actions undertaken by the neural policy.

**Synthesis.** Reducing a complex neural policy to a simpler yet safe deterministic program is possible because we do not require other properties from the oracle; specifically, we do not require that the deterministic program precisely mirrors the performance of the neural policy. For example, experiments described in [35] show that while a linear-policy network, we posit that a linear reduction can be sufficiently expressive to describe necessary safety constraints. Based on this hypothesis, for our inverted pendulum example, we can explore a linear program space from which a deterministic program \( \mathcal{P}_\theta \) can be drawn expressed in terms of the following program sketch:

```
def \( \mathcal{P}(\theta_1, \theta_2)(\eta, \omega) \): return \( \theta_1 \eta + \theta_2 \omega \)
```

Here, \( \theta = [\theta_1, \theta_2] \) are unknown parameters that need to be synthesized. Intuitively, the program weights the importance of \( \eta \) and \( \omega \) at a particular state to provide a feedback control action to mitigate the deviation of the inverted pendulum from \( (\eta = 0^\circ, \omega = 0^\circ) \).

Our search-based synthesis sets \( \theta \) to 0 initially. It runs the deterministic program \( \mathcal{P}_\theta \) instead of the oracle neural policy \( \pi_w \) within the environment \( C \) defined in Fig. 1 (in this case, the state transition system represents the differential equation specification of the controller) to collect a batch of trajectories. A run of the state transition system of \( C[\mathcal{P}_\theta] \) produces a finite trajectory \( s_0, s_1, \ldots, s_T \). We find \( \theta \) from the following optimization task that realizes (1):

\[
\max_{\theta \in \mathbb{R}^2} \mathbb{E}[\sum_{t=0}^{T} d(\pi_w, \mathcal{P}_\theta, s_t)]
\]

(2)

where \( d(\pi_w, \mathcal{P}_\theta, s_t) \equiv \begin{cases} (\mathcal{P}_\theta(s_t) - \pi_w(s_t))^2 & s_t \notin S_u \\ -\max & s_t \in S_u \end{cases} \). This equation aims to search for a program \( \mathcal{P}_\theta \) at minimal distance from the neural oracle \( \pi_w \) along sampled trajectories, while simultaneously maximizing the likelihood that \( \mathcal{P}_\theta \) is safe.

Our synthesis procedure described in Sec. 4.1 is a random search-based optimization algorithm [30]. We sample a new position of \( \theta \) iteratively from its hypersphere of a given small radius surrounding the current position of \( \theta \) and move to the new position (w.r.t. a learning rate) as dictated by Equation (2). For the running example, our search synthesizes:

```
def \( \mathcal{P}(\eta, \omega) \): return -12.05\eta + 5.87\omega
```

The synthesized program can be used to intuitively interpret how the neural oracle works. For example, if a pendulum with a positive angle \( \eta > 0 \) leans towards the right (\( \omega > 0 \)), the controller will need to generate a large negative control action to force the pendulum to move left.

**Verification.** Since our method synthesizes a deterministic program \( \mathcal{P} \), we can leverage off-the-shelf formal verification algorithms to verify its safety with respect to the state transition system \( C \) defined in Fig. 1. To ensure that \( \mathcal{P} \) is safe, we must ascertain that it can never transition to an unsafe state, i.e., a state that causes the pendulum to fall. When framed as a formal verification problem, answering such a question is tantamount to discovering an inductive invariant \( \varphi \) that represents all safe states over the state transition system:

1. **Safe:** \( \varphi \) is disjoint with all unsafe states \( S_u \).
2. **Init:** \( \varphi \) includes all initial states \( S_0 \).
3. **Induction:** Any state in \( \varphi \) transitions to another state in \( \varphi \) and hence cannot reach an unsafe state.

Inspired by template-based constraint solving approaches on inductive invariant inference [19, 20, 24, 34], the verification algorithm described in Sec. 4.2 uses a constraint solver to look for an inductive invariant in the form of a convex barrier certificate [34] \( E(s) \leq 0 \) that maps all the states in the (safe) reachable set to non-positive reals and all the states in the unsafe set to positive reals. The basic idea is to identify a polynomial function \( E : \mathbb{R}^n \rightarrow \mathbb{R} \) such that 1) \( E(s) > 0 \) for any state \( s \in S_u \), 2) \( E(s) \leq 0 \) for any state \( s \in S_0 \), and 3) \( E(s') - E(s) \leq 0 \) for any state \( s \) that transitions to \( s' \) in the state transition system \( C[\mathcal{P}] \). The second and third condition collectively guarantee that \( E(s) \leq 0 \) for any state \( s \) in the reachable set, thus implying that an unsafe state in \( S_u \) can never be reached.

Fig. 3(a) draws the discovered invariant in blue for \( C[\mathcal{P}] \) given the initial and unsafe states where \( \mathcal{P} \) is the synthesized program for the inverted pendulum system. We can conclude that the safety property is satisfied by the \( \mathcal{P} \) controlled system as all reachable states do not overlap with unsafe states. In case verification fails, our approach conducts a counterexample-guided loop (Sec. 4.2) to iteratively synthesize safe deterministic programs until verification succeeds.

**Shielding.** Keep in mind that a safety proof of a reduced deterministic program of a neural network does not automatically lift to a safety argument of the neural network from which it was derived since the network may exhibit behaviors not fully captured by the simpler deterministic program. To bridge this divide, we propose to recover soundness at runtime by monitoring system behaviors of a neural network in its actual environment (deployment) context.
We find that the neural policy is never interrupted by the shield derived from the program will not frequently interrupt the neural network’s decision, allowing the combined system to perform (close-to) optimally.

Increasing safety with a shield. The shield S can serve as an oracle, it is expected that the shield derived from the program will not frequently interrupt the neural network’s decision, allowing the combined system to perform (close-to) optimally.

In our running example, suppose we wish to operate the inverted pendulum in an environment such as a Segway transporter in which the model is prohibited from swinging significantly and whose angular velocity must be suitably restricted. We might specify the following new constraints on state parameters to enforce these conditions:

\[ S_u : \{ (\eta, \omega) | \sim(-30^\circ < \eta < 30^\circ \land -30^\circ < \omega < 30^\circ) \} \]

Because of the dependence of a neural network to the quality of training data used to build it, environment changes that deviate from assumptions made at training-time could result in a costly retraining exercise because the network must learn a new way to penalize unsafe actions that were previously safe. However, training a neural network from scratch requires substantial non-trivial effort, involving fine-grained tuning of training parameters or even network structures.

In our framework, the existing network provides a reasonable approximation to the desired behavior. To incorporate the additional constraints defined by the new environment C’, we attempt to synthesize a new deterministic program P’ for C’, a task based on our experience is substantially easier to achieve than training a new neural network policy from scratch. This new program can be used to protect the original network provided that we can use the aforementioned verification approach to formally verify that C’[P’] is safe by identifying a new inductive invariant φ’. As depicted in Fig. 4, we simply build a new shield S’ that is composed of the program P’ and the safety boundary captured by φ’.

The shield S’ can ensure the safety of the neural network in the environment context C’ with a strengthened safety condition, despite the fact that the neural network was trained in a different environment context C.

Fig. 3(b) depicts the new unsafe states in C’ (colored in red). It extends the unsafe range draw in Fig. 3(a) so the inductive invariant learned there is unsafe for the new one. Our approach synthesizes a new deterministic program for which we learn a more restricted inductive invariant depicted in green in Fig. 3(b). To characterize the effectiveness of our shielding approach, we examined 1000 simulation trajectories of this restricted version of the inverted pendulum system, each of which is comprised of 5000 simulation steps with the safety constraints defined by C’.

Without the

Fig. 4. The Framework of Neural Network Shielding.

Fig. 3. Invariant Inference on Inverted Pendulum.
We model the context with the assumption that a control action \( t \in T \) to interrupt the neural network’s decision was extremely low. Out of a total of 5000x1000 decisions, we only observed 414 instances (0.00828%) where the shield interfered with (i.e., overrode) the decision made by the network.

3 Problem Setup

We model the context \( C \) of a control policy as an environment state transition system \( C[\cdot] = (X, \mathcal{A}, S, S_0, S_n, T_\pi[\cdot], f, r) \). Note that \( \cdot \) is intentionally left open to deploy neural control policies. Here, \( X \) is a finite set of variables interpreted over the reals \( \mathbb{R} \) and \( S = \mathbb{R}^N \) is the set of all valuations of the variables \( X \). We denote \( s \in S \) to be an \( n \)-dimensional environment state and \( a \in \mathcal{A} \) to be a control action where \( \mathcal{A} \) is an infinite set of \( m \)-dimensional continuous actions that a learning-enabled controller can perform. We use \( S_0 \in S \) to specify a set of initial environment states and \( S_n \in S \) to specify a set of unsafe environment states that a safe controller should avoid. The transition relation \( T_\pi[\cdot] \) defines how one state transitions to another given an action by an unknown policy. We assume that \( T_\pi[\cdot] \) is governed by a standard differential equation \( f \) defining the relationship between a continuously varying state \( s(t) \) and action \( a(t) \) and its rate of change \( \dot{s}(t) \) over time \( t \):

\[
\dot{s}(t) = f(s(t), a(t))
\]

We assume \( f \) is defined by equations of the form \( \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n \) such as the example in Fig. 1. In the following, we often omit the time variable \( t \) for simplicity. A deterministic neural network control policy \( \pi_w : \mathbb{R}^n \to \mathbb{R}^m \) parameterized by a set of weight values \( w \) is a function mapping an environment state \( s \) to a control action \( a \) where

\[
a = \pi_w(s)
\]

By providing a feedback action, a policy can alter the rate of change of state variables to realize optimal system control.

The transition relation \( T_\pi[\cdot] \) is parameterized by a control policy \( \pi \) that is deployed in \( C \). We explicitly model this deployment as \( T_\pi[\pi] \). Given a control policy \( \pi_w \), we use \( T_\pi[\pi_w] : S \times S \) to specify all possible state transitions allowed by the policy. We assume that a system transitions in discrete time instants \( kT \) where \( k = 0, 1, 2, \cdots \) and \( t \) is a fixed time step \( (t > 0) \). A state \( s \) transitions to a next state \( s' \) after time \( t \) with the assumption that a control action \( a(r) \) at time \( r \) is a constant between the time period \( r \in [0, t) \). Using Euler’s method\(^2\), we discretize the continuous dynamics \( f \) with finite difference approximation so it can be used in the discretized transition relation \( T_{\pi}[\pi_w] \). Then \( T_{\pi}[\pi_w] \) can compute these estimates by the following difference equation:

\[
T_{\pi}[\pi_w] := \{(s, s') \mid s' = s + f(s, \pi_w(s)) \times t\}
\]

Environment Disturbance. Our model allows bounded external properties (e.g., additional environment-imposed constraints) by extending the definition of change of rate: \( s = f(s, a) + d \) where \( d \) is a vector of random disturbances. We use \( d \) to encode environment disturbances in terms of bounded nondeterministic values. We assume that tight upper and lower bounds of \( d \) can be accurately estimated at runtime using multivariate normal distribution fitting methods.

Trajectory. A trajectory \( h \) of a state transition system \( C[\pi_w] \) which we denote as \( h \in C[\pi_w] \) is a sequence of states \( s_0, \cdots, s_t, s_{t+1}, \cdots \) where \( s_0 \in S_0 \) and \( (s_i, s_{i+1}) \in T_{\pi}[\pi_w] \) for all \( i \geq 0 \). We use \( C \subseteq C[\pi_w] \) to denote a set of trajectories. The reward that a control policy receives on performing an action \( a \) in a state \( s \) is given by the reward function \( r(s, a) \).

Reinforcement Learning. The goal of reinforcement learning is to maximize the reward that can be collected by a neural control policy in a given environment context \( C \). Such problems can be abstractly formulated as

\[
\max_{w \in \mathbb{R}} \mathbb{E}[r(\pi_w)] = \mathbb{E}[r(\pi_w)]
\]

Assume that \( s_0, s_1, \ldots, s_T \) is a trajectory of length \( T \) of the state transition system and \( r(\pi_w) = \lim_{T \to \infty} \sum_{k=0}^{T} r(s_k, \pi_w(s_k)) \) is the cumulative reward achieved by the policy \( \pi_w \) from this trajectory. Thus this formulation uses simulations of the transition system with finite length rollouts to estimate the expected cumulative reward collected over \( T \) time steps and aim to maximize this reward. Reinforcement learning assumes policies are expressible in some executable structure (e.g. a neural network) and allows samples generated from one policy to influence the estimates made for others. The two main approaches for reinforcement learning are value function estimation and direct policy search. We refer readers to [27] for an overview.

Safety Verification. Since reinforcement learning only considers finite length rollouts, we wish to determine if a control policy is safe to use under an infinite time horizon. Given a state transition system, the safety verification problem is concerned with verifying that no trajectories contained in \( S \) starting from an initial state in \( S_0 \) reach an unsafe state in \( S_u \). However, a neural network is a representative of a class of deep and sophisticated models that challenges the capability of the state-of-the-art verification techniques. This level of complexity is exacerbated in our work because we consider the long term safety of a neural policy deployed within a nontrivial environment context \( C \) that in turn is described by a complex infinite-state transition system.
E ::= v | x | Φ(E₁, . . . , Eₖ)
φ ::= E ≤ 0
P ::= return E | if φ then return E else P

Figure 5. Syntax of the Policy Programming Language.

4 Verification Procedure

To verify a neural network control policy πₜ with respect to
an environment context C, we first synthesize a deterministic policy program P from the neural policy. We require that P both (a) broadly resembles its neural oracle and (b) additionally satisfies a desired safety property when it is deployed in C. We conjecture that a safety proof of C[P] is easier to construct than that of C[πₜ]. More importantly, we leverage the safety proof of C[P] to ensure the safety of C[πₜ].

4.1 Synthesis

Fig. 5 defines a search space for a deterministic policy program P where E and φ represent the basic syntax of (polynomial) program expressions and inductive invariants, respectively. Here v ranges over a universe of numerical constants, x represents variables, and Φ is a basic operator including + and ×. A deterministic program P also features conditional statements using φ as branching predicates.

We allow the user to define a sketch [41, 42] to describe the shape of target policy programs using the grammar in Fig. 5. We use P[θ] to represent a sketch where θ represents unknowns that need to be filled-in by the synthesis procedure. We use P₀ to represent a synthesized program with known values of θ. Similarly, the user can define a sketch of an inductive invariant φ[.] that is used (in Sec. 4.2) to learn a safety proof to verify a synthesized program P₀.

We do not require the user to explicitly define conditional statements in a program sketch. Our synthesis algorithm uses verification counterexamples to lazily add branch predicates φ under which a program performs different computations depending on whether φ evaluates to true or false. The end user simply writes a sketch over basic expressions. For example, a sketch that defines a family of linear function over a collection of variables can be expressed as:

P[θ](X) ::= return θ₁x₁ + θ₂x₂ + . . . + θₙxn + θₙ₊₁

Here X = (x₁, x₂, . . . ) are system variables in which the coefficient constants θ = (θ₁, θ₂, . . . ) are unknown.

The goal of our synthesis algorithm is to find unknown values of θ that maximize the likelihood that P₀ resembles the neural oracle πₜ while still being a safe program with respect to the environment context C:

θ* = arg maxθ∈Rₙ₊₁ d(πₜ, P₀, C)

where d measures the distance between the outputs of the estimate program P₀ and the neural policy πₜ subject to safety constraints. To avoid the computational complexity that arises if we consider a solution to this equation analytically as an optimization problem, we instead approximate d(πₜ, P₀, C) by randomly sampling a set of trajectories C that are encountered by P₀ in the environment state transition system C[P₀]:

d(πₜ, P₀, C) ≈ d(πₜ, P₀, C) s.t. C ⊆ C[P₀]

We estimate θ* using these sampled trajectories C and define:

d(πₜ, P₀, h) = \sum_{h∈C} d(πₜ, P₀, h)

Since each trajectory h ∈ C is a finite rollout s₀, . . . , s₉ of length T, we have:

d(πₜ, P₀, h) = \sum_{t=0}^{T} \max_{s₁ \in S_u} \| (P₀(s₁) − πₜ(s₁)) \| \quad s₁ \notin S_u

where \| · \| is a suitable norm. As illustrated in Sec. 2.2, we aim to minimize the distance between a synthesized program P₀ from a sketch space and its neural oracle along sampled trajectories encountered by P₀ in the environment context but put a large penalty on states that are unsafe.

Random Search. We implement the idea encoded in equation (5) in Algorithm 1 that depicts the pseudocode of our policy interpretation approach. It takes as inputs a neural policy, a policy program sketch parameterized by θ, and an environment state transition system and outputs a synthesized policy program P₀.

An efficient way to solve equation (5) is to directly perturb θ in the search space by adding random noise and then update θ based on the effect on this perturbation [30]. We choose a direction uniformly at random on the sphere in parameter space, and then optimize the goal function along this direction. To this end, in line 3 of Algorithm 1, we sample Gaussian noise δ to be added to policy parameters θ in both directions where v is a small positive real number. In line 4 and line 5, we sample trajectories C₁ and C₂ from the state transition systems obtained by running the perturbed policy program P₀⁺δ and P₀⁻δ in the environment context C.

We evaluate the proximity of these two policy programs to the neural oracle and, in line 6 of Algorithm 1, to improve the

Algorithm 1: Synthesize (πₜ, P[θ], C)::

1. θ ← 0;
2. do
3.  δ ← Sample δ from a zero mean Gaussian vector;
4.  Sample a set of trajectories C₁ using C[P₀⁺δ];
5.  Sample a set of trajectories C₂ using C[P₀⁻δ];
6.  θ ← θ + α [d(πₜ, P₀⁺δ, C₁) − d(πₜ, P₀⁻δ, C₂)] δ;
7.  while θ is not converged;
8.  return P₀
policy program, we also optimize equation (5) by updating $\theta$ with a finite difference approximation along the direction:

$$\theta \leftarrow \theta + \alpha \left[ \frac{d(P_{\theta+\delta\eta}, \pi_\omega, C_1) - d(P_{\theta-\delta\eta}, \pi_\omega, C_2)}{\delta} \right]$$  \hspace{1cm} (6)

where $\alpha$ is a predefined learning rate. Such an update increment corresponds to an unbiased estimator of the gradient of $\theta$ [29, 33] for equation (5). The algorithm iteratively updates $\theta$ until convergence.

4.2 Verification

A synthesized policy program $P$ is verified with respect to an environment context given as an infinite state transition system defined in Sec. 3: $C[P] = (X, A, S, S_0, S_u, T_r[P], f, r)$. Our verification algorithm learns an inductive invariant $\varphi$ over the transition relation $T_r[P]$ formally proving that all possible system behaviors are encapsulated in $\varphi$ and $\varphi$ is required to be disjoint with all unsafe states $S_u$.

The sketch includes all monomials over $\eta$ whose degrees are no greater than 4. The coefficients $c = [c_0, \cdots, c_p]$ are used to parameterize $E[c]$ in an affine manner:

$$E[c](X) = \sum_{i=0}^{p} c_i b_i(X)$$

where the $b_i(X)$’s are some monomials in variables $X$. As a heuristic, the user can simply determine an upper bound on the degree of $E[c]$, and then include all monomials whose degrees are no greater than the bound in the sketch. Large values of the bound enable verification of more complex safety conditions, but impose greater demands on the constraint solver; small values capture coarser safety properties, but are easier to solve.

Example 4.1. Consider the inverted pendulum system in Sec. 2.2. To discover an inductive invariant for the transition system, the user might choose to define an upper bound of 4, which results in the following polynomial invariant sketch:

$$\varphi[c](\eta, \omega) := E[c](\eta, \omega) \leq 0$$

where $E[c](\eta, \omega) = c_0 \eta^4 + c_1 \eta^3 \omega + c_2 \eta^2 \omega^2 + c_3 \eta \omega^3 + c_4 \omega^4 + c_5 \eta^3 + \cdots + c_p$

The sketch includes all monomials over $\eta$ and $\omega$, whose degrees are no greater than 4. The coefficients $c = [c_0, \cdots, c_p]$ are unknown and need to be synthesized.

To synthesize these unknowns, we require that $E[c]$ must satisfy the following verification conditions:

$$\forall (s) \in S_u \quad E[c](s) > 0 \quad (8)$$

$$\forall (s) \in S_0 \quad E[c](s) \leq 0 \quad (9)$$

$$\forall (s, s') \in T_r[P] \quad E[c](s') - E[c](s) \leq 0 \quad (10)$$

We claim that $\varphi := E[c](x) \leq 0$ defines an inductive invariant because verification condition (9) ensures that any initial state $s_0 \in S_0$ satisfies $\varphi$ since $E[c](s_0) \leq 0$; verification condition (10) asserts that along the transition from a state $s \in \varphi$ (so $E[c](s) \leq 0$) to a resulting state $s'$, $E[c]$ cannot become positive so $s'$ satisfies $\varphi$ as well. Finally, according to verification condition (8), $\varphi$ does not include any unsafe state $s_u \in S_u$ as $E[c](s_u)$ is positive.

Verification conditions (8) (9) (10) are polynomials over reals. Synthesizing unknown coefficients can be left to an SMT solver [13] after universal quantifiers are eliminated using a variant of Farkas Lemma as in [20]. However, observe that $E[c]$ is convex. We can gain efficiency by finding unknown coefficients $c$ using off-the-shelf convex constraint solvers following [34]. Encoding verification conditions (8) (9) (10) as polynomial inequalities, we search $c$ that can prove non-negativity of these constraints via an efficient and convex sum of squares programming solver [26]. Additional technical details are provided in the supplementary material [49].

Counterexample-guided Inductive Synthesis (CEGIS).

Given the environment state transition system $C[P]$ deployed with a synthesized program $P$, the verification approach above can compute an inductive invariant over the state transition system or tell if there is no feasible solution in the given set of candidates. Note however that the latter does not necessarily imply that the system is unsafe.

Since our goal is to learn a safe deterministic program from a neural network, we develop a counterexample guided inductive synthesis approach. A CEGIS algorithm in our context is challenging because safety verification is necessarily incomplete, and may not be able to produce a counterexample that serves as an explanation for why a verification attempt is unsuccessful.

We solve the incompleteness challenge by leveraging the fact that we can simultaneously synthesize and verify a program. Our CEGIS approach is given in Algorithm 2. A counterexample is an initial state on which our synthesized program is not yet proved safe. Driven by these counterexamples, our algorithm synthesizes a set of programs from a sketch along with the conditions under which we can switch from one synthesized program to another.

Algorithm 2 takes as input a neural policy $\pi_\omega$, a program sketch $P[\theta]$ and an environment context $C$. It maintains synthesized policy programs in policies in line 1, each of which is inductively verified safe in a partition of the universe state

---

1For arbitrary $E_1(x)$ and $E_2(x)$ satisfying the verification conditions and $\alpha \in [0, 1]$, $E(x) = \alpha E_1(x) + (1 - \alpha)E_2(x)$ satisfies the conditions as well.
Algorithm 2: CEGIS ($\pi_w, \mathcal{P}[\theta], C[:])$

1. policies ← $\emptyset$
2. covers ← $\emptyset$
3. while $C.S_0 \not\subseteq$ covers do
   4. search $s_0$ such that $s_0 \in C.S_0 \land s_0 \not\in$ covers;
   5. $r^* \leftarrow$ Diameter($C.S_0$);
   6. do
      7. $\phi_{bound} \leftarrow \{s \mid s \in (s_0 - r^*, s_0 + r^*)\}$;
      8. $\bar{C} \leftarrow C$ where $\bar{C}.S_0 = (C.S_0 \cap \phi_{bound})$;
      9. $\theta \leftarrow$ Synthesize($\pi_w, \mathcal{P}[\theta], \bar{C}[:]$);
      10. $\varphi \leftarrow$ Verify($\bar{C}[\theta]$);
      11. if $\varphi$ is False then
          12. $r^* \leftarrow r^*/2$;
          13. else
          14. covers ← covers $\cup$ $\{s \mid \varphi(s)\}$;
          15. policies ← policies $\cup$ $\{\mathcal{P}[\theta, \varphi]\}$;
          16. break;
   17. end
4. while True;
18. return policies

Figure 6. CEGIS for Verifiable Reinforcement Learning.

The entire initial state space is covered (line 3 to 18). The output of Algorithm 2 is $[\{\mathcal{P}_{\theta_1, \varphi_1}, \mathcal{P}_{\theta_2, \varphi_2}, \cdots\}]$ that essentially defines conditional statements in a synthesized program performing different actions depending on whether a specified invariant condition evaluates to true or false.

Theorem 4.2. If $\text{CEGIS}(\pi_w, \mathcal{P}[\theta], C[:]) = [\{\mathcal{P}_{\theta_1, \varphi_1}, \mathcal{P}_{\theta_2, \varphi_2}, \cdots\}]$ (as defined in Algorithm 2), then the deterministic program $\mathcal{P}$:

$$\forall x. \text{if } \varphi_1(x): \text{return } \mathcal{P}_{\theta_1}(x) \text{ else if } \varphi_2(x): \text{return } \mathcal{P}_{\theta_2}(x) \cdots$$

is safe in the environment $C$ meaning that $\varphi_1 \lor \varphi_2 \lor \cdots$ is an inductive invariant of $C[\mathcal{P}]$ proving that $C.S_u$ is unreachable.

Example 4.3. We illustrate the proposed counterexample guided inductive synthesis method by means of a simple example, the Duffing oscillator [22], a nonlinear second-order environment. The transition relation of the environment system $C$ is described with the differential equation:

$$\dot{x} = y$$

$$\dot{y} = -0.6y - x - x^3 + a$$

where $x, y$ are the state variables and $a$ the continuous control action given by a well-trained neural feedback control policy $\pi$ such that $a = \pi(x, y)$. The control objective is to regulate the state to the origin from a set of initial states $C.S_0 : \{x, y\} \in [-5, 5] \land -2 \leq y \leq 2$. To be safe, the controller must be able to avoid a set of unsafe states $C.S_u : \{x, y\} \not\in [-5, 5] \land -5 \leq y \leq 5$.

Given a program sketch as in the pendulum example, that is $\mathcal{P}[\theta](x, y) := \theta_1x + \theta_2y$, the user can ask the constraint solver to reason over a small (say 4) order polynomial invariant sketch for a synthesized program as in Example 4.1.

Algorithm 2 initially samples an initial state $s$ as $\{x = -0.46, y = -0.36\}$ from $S_0$. The inner do-while loop of the algorithm can discover a sub-region of initial states in the dotted box of Fig. 6(a) that can be leveraged to synthesize a verified deterministic policy $\mathcal{P}_1(x, y) := 0.39x - 1.41y$ from the sketch. We also obtain an inductive invariant showing that the synthesized policy can always maintain the controller within the invariant set drawn in purple in Fig. 6(a):
A safety proof of a synthesized deterministic program \( \varphi \) can guarantee stability in the supplementary material [49].

This invariant explains why the initial state space used for verification does not include the entire \( C_S \); a counterexample initial state \( s' = (x = 2.249, y = 2) \) is not covered by the invariant for which the synthesized policy program above is not verified safe. The CEGIS loop in line 3 of Algorithm 2 uses \( s' \) to synthesize another deterministic policy \( P_2(x, y) := 0.88x - 2.34y \) from the sketch whose learned inductive invariant is depicted in blue in Fig. (2):

\[
\begin{align*}
\varphi_2 &\equiv 12.8x^4 + 0.9x^3y - 0.2x^2y^2 - 5.9x^3 - 1.5xy^2 - 0.3y^3 + 2.2x^2 + 4.7xy + 40.4y^2 - 619 \leq 0. \tag{2} \\
\end{align*}
\]

Algorithm 2 then terminates because \( \varphi_1 \lor \varphi_2 \) covers \( C_S \). Our system interprets the two synthesized deterministic policies as the following deterministic program \( P_{oscillator} \) using the syntax in Fig. 5:

```
def P_{oscillator}(x, y):
    if 20.9x^4 + 2.9x^3y + 1.4x^2y^2 + 0.4xy^3 + 29.6x^3 + 20.1x^2y + 11.3xy^2 + 1.6y^3 + 25.2x^2 + 39.2xy + 53.7y^2 - 680 \leq 0:
        return 0.39x - 1.4y
    else if 12.8x^4 + 0.9x^3y - 0.2x^2y^2 - 5.9x^3 - 1.5xy^2 - 0.3y^3 + 2.2x^2 + 4.7xy + 40.4y^2 - 619 \leq 0:
        return 0.88x - 2.34y
    else abort # unsafe
```

Neither of the two deterministic policies enforce safety by themselves on all initial states but do guarantee safety when combined together because by construction, \( \varphi = \varphi_1 \lor \varphi_2 \) is an inductive invariant of \( P_{oscillator} \) in the environment \( C \).

Although Algorithm 2 is sound, it may not terminate as \( r^* \) in the algorithm can become arbitrarily small and there is also no restriction on the size of potential counterexamples. Nonetheless, our experimental results indicate that the algorithm performs well in practice.

### 4.3 Shielding

A safety proof of a synthesized deterministic program of a neural network does not automatically lift to a safety argument of the neural network from which it was derived since the network may exhibit behaviors not captured by the simpler deterministic program. To bridge this divide, we recover soundness at runtime by monitoring system behaviors of a neural network in its environment context by using the synthesized policy program and its inductive invariant as a shield. The pseudo-code for using a shield is given in Algorithm 3. In line 1 of Algorithm 3, for a current state \( s \), we use the state transition system of our environment context model to predict the next state \( s' \). If \( s' \) is not within \( \varphi \), we are unsure whether entering \( s' \) would inevitably make the system unsafe as we lack a proof that the neural oracle is safe. However, we do have a guarantee that if we follow the synthesized program \( P \), the system would stay within the safety boundary defined by \( \varphi \) that Algorithm 2 has formally proved. We do so in line 3 of Algorithm 3, using \( P \) and \( \varphi \) as shields, intervening only if necessary so as to restrict the shield from unnecessarily intervening the neural policy. \(^4\)

\(^4\) We also extended our approach to synthesize deterministic programs which can guarantee stability in the supplementary material [49].

### 5 Experimental Results

We have applied our framework on a number of challenging control- and cyberphysical-system benchmarks. We consider the utility of our approach for verifying the safety of trained neural network controllers. We use the deep policy gradient algorithm [28] for neural network training, the Z3 SMT solver [13] to check convergence of the CEGIS loop, and the Mosk constraint solver [6] to generate inductive invariants of synthesized programs from a sketch. All of our benchmarks are verified using the program sketch defined in equation (4) and the invariant sketch defined in equation (7).

We report simulation results on our benchmarks over 1000 runs (each run consists of 5000 simulation steps). Each simulation time step is fixed 0.01 second. Our experiments were conducted on a standard desktop machine consisting of Intel(R) Core(TM) i7-8700 CPU cores and 64GB memory.

#### Case Study on Inverted Pendulum

We first give more details on the evaluation result of our running example, the inverted pendulum. Here we consider a more restricted safety condition that deems the system to be unsafe when the pendulum’s angle is more than 23° from the origin (i.e., significant swings are further prohibited). The controller is a 2-hidden-layers neural model (240 × 200). Our tool interprets the neural network as a program containing three conditional branches:

#### Over 1000 simulation runs, we found 60 unsafe cases when running the neural controller alone. Importantly, running the neural controller in tandem with the above verified and synthesized program can prevent all unsafe neural decisions. There were only with 65 interventions from the program on the neural network. Our results demonstrate that CEGIS is important to ensure a synthesized program is safe to use. We report more details including synthesis times below.

One may ask why we do not directly learn a deterministic program to control the device (without appealing to the neural policy at all), using reinforcement learning to synthesize its unknown parameters. Even for an example as simple as
the inverted pendulum, our discussion above shows that it is difficult (if not impossible) to derive a single straight-line (linear) program that is safe to control the system. Even for scenarios in which a straight-line program suffices, using existing RL methods to directly learn unknown parameters in our sketches may still fail to obtain a safe program. For example, we considered if a linear control policy can be learned to (1) prevent the pendulum from falling down (2) and require a pendulum control action to be strictly within the range $[-1, 1]$ (e.g., operating the controller in an environment with low power constraints). We found that despite many experiments on tuning learning rates and rewards, directly training a linear control program to conform to this restriction with either reinforcement learning (e.g. policy gradient) or random search [29] was unsuccessful because of undesirable overfitting. In contrast, neural networks work much better for these RL algorithms and can be used to guide the synthesis of a deterministic program policy. Indeed, by treating the neural policy as an oracle, we were able to quickly discover a straight-line linear deterministic program that in fact satisfies this additional motion constraint.

Safety Verification. Our verification results are given in Table 1. In the table, \textbf{Vars} represents the number of variables in a control system - this number serves as a proxy for application complexity; \textbf{Size} the number of neurons in hidden layers of the network; the \textbf{Training} time for the network; and, its \textbf{Failures}, the number of times the network failed to satisfy the safety property in simulation. The table also gives the \textbf{Size} of a synthesized program in term of the number of polices found by Algorithm 2 (used to generate conditional statements in the program); its \textbf{Synthesis} time; the \textbf{Overhead} of our approach in terms of the additional cost (compared to the non-shielded variant) in running time to use a shield; and, the number of \textbf{Interventions}, the number of times the shield was invoked across all simulation runs. We also report performance gap between of a shield neural policy (\textbf{NN}) and a purely programmatic policy (\textbf{Program}), in terms of the number of steps on average that a controlled system spends in reaching a steady state of the system (i.e., a convergence state).

The first five benchmarks are linear time-invariant control systems adapted from [15]. The safety property is that the reach set has to be within a safe rectangle. Benchmark Biology defines a minimal model of glycemic control in diabetic patients such that the dynamics of glucose and insulin interaction in the blood system are defined by polynomials [10]. For safety, we verify that the neural controller ensures that the level of plasma glucose concentration is above a certain threshold. Benchmark DataCenter Cooling is a model of a collection of three server racks each with their own cooling devices and they also shed heat to their neighbors. The safety property is that a learned controller must keep the data center below a certain temperature. In these benchmarks, the cost to query the network oracle constitutes the dominant time for generating the safety shield in these benchmarks. Given the simplicity of these benchmarks, the neural network controllers did not violate the safety condition in our trials, and moreover there were no interventions from the safety shields that affected performance.

The next three benchmarks Quadcopter, (Inverted) Pendulum and Cartpole are selected from classic control applications and have more sophisticated safety conditions. We have discussed the inverted pendulum example at length earlier. The Quadcopter environment tests whether a controlled quadcopter can realize stable flight. The environment of Cartpole consists of a pole attached to an unactuated joint connected to a cart that moves along a frictionless track. The system is unsafe when the pole’s angle is more than

### Table 1. Experimental Results on Deterministic Program Synthesis, Verification, and Shielding.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Vars</th>
<th>Neural Network Size</th>
<th>Training Time</th>
<th>Failures</th>
<th>Deterministic Program as Shield Size</th>
<th>Synthesis Time</th>
<th>Overhead</th>
<th>Interventions</th>
<th>Performance Gap</th>
<th>NN</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satellite</td>
<td>2</td>
<td>$240 \times 200$</td>
<td>957s</td>
<td>0</td>
<td>1</td>
<td>160s</td>
<td>3.37%</td>
<td>0</td>
<td>5.7</td>
<td>9.7</td>
<td></td>
</tr>
<tr>
<td>DCMotor</td>
<td>3</td>
<td>$240 \times 200$</td>
<td>944s</td>
<td>0</td>
<td>1</td>
<td>68s</td>
<td>2.03%</td>
<td>0</td>
<td>11.9</td>
<td>12.2</td>
<td></td>
</tr>
<tr>
<td>Tape</td>
<td>3</td>
<td>$240 \times 200$</td>
<td>980s</td>
<td>0</td>
<td>1</td>
<td>42s</td>
<td>2.63%</td>
<td>0</td>
<td>3.0</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>Magnetic Pointer</td>
<td>3</td>
<td>$240 \times 200$</td>
<td>992s</td>
<td>0</td>
<td>1</td>
<td>85s</td>
<td>2.92%</td>
<td>0</td>
<td>8.3</td>
<td>8.8</td>
<td></td>
</tr>
<tr>
<td>Suspension</td>
<td>4</td>
<td>$240 \times 200$</td>
<td>960s</td>
<td>0</td>
<td>1</td>
<td>41s</td>
<td>8.71%</td>
<td>0</td>
<td>4.7</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>Biology</td>
<td>3</td>
<td>$240 \times 200$</td>
<td>978s</td>
<td>0</td>
<td>1</td>
<td>168s</td>
<td>5.23%</td>
<td>0</td>
<td>2464</td>
<td>2599</td>
<td></td>
</tr>
<tr>
<td>DataCenter Cooling</td>
<td>3</td>
<td>$240 \times 200$</td>
<td>968s</td>
<td>0</td>
<td>1</td>
<td>168s</td>
<td>4.69%</td>
<td>0</td>
<td>14.6</td>
<td>40.1</td>
<td></td>
</tr>
<tr>
<td>Quadcopter</td>
<td>2</td>
<td>$300 \times 200$</td>
<td>990s</td>
<td>182</td>
<td>2</td>
<td>67s</td>
<td>6.41%</td>
<td>185</td>
<td>7.2</td>
<td>9.8</td>
<td></td>
</tr>
<tr>
<td>Pendulum</td>
<td>2</td>
<td>$240 \times 200$</td>
<td>962s</td>
<td>60</td>
<td>3</td>
<td>1107s</td>
<td>9.65%</td>
<td>65</td>
<td>44.2</td>
<td>58.6</td>
<td></td>
</tr>
<tr>
<td>CartPole</td>
<td>4</td>
<td>$300 \times 200$</td>
<td>990s</td>
<td>47</td>
<td>4</td>
<td>998s</td>
<td>5.62%</td>
<td>1799</td>
<td>681.3</td>
<td>1912.6</td>
<td></td>
</tr>
<tr>
<td>Self-Driving</td>
<td>4</td>
<td>$300 \times 200$</td>
<td>990s</td>
<td>61</td>
<td>1</td>
<td>185s</td>
<td>4.66%</td>
<td>236</td>
<td>145.9</td>
<td>513.6</td>
<td></td>
</tr>
<tr>
<td>Lane Keeping</td>
<td>4</td>
<td>$240 \times 200$</td>
<td>895s</td>
<td>36</td>
<td>1</td>
<td>183s</td>
<td>8.65%</td>
<td>64</td>
<td>375.3</td>
<td>643.5</td>
<td></td>
</tr>
<tr>
<td>4-Car platoon</td>
<td>8</td>
<td>$500 \times 400 \times 300$</td>
<td>1160s</td>
<td>8</td>
<td>4</td>
<td>609s</td>
<td>3.17%</td>
<td>8</td>
<td>7.6</td>
<td>9.6</td>
<td></td>
</tr>
<tr>
<td>8-Car platoon</td>
<td>16</td>
<td>$500 \times 400 \times 300$</td>
<td>1165s</td>
<td>40</td>
<td>1</td>
<td>1217s</td>
<td>6.05%</td>
<td>1080</td>
<td>38.5</td>
<td>55.4</td>
<td></td>
</tr>
<tr>
<td>Oscillator</td>
<td>18</td>
<td>$240 \times 200$</td>
<td>1023s</td>
<td>371</td>
<td>1</td>
<td>618s</td>
<td>21.31%</td>
<td>93703</td>
<td>693.5</td>
<td>1135.3</td>
<td></td>
</tr>
</tbody>
</table>
30° from being upright or the cart moves by more than 0.3 meters from the origin. We observed safety violations in each of these benchmarks that were eliminated using our verification methodology. Notably, the number of interventions is remarkably low, as a percentage of the overall number of simulation steps.

Benchmark Self-driving defines a single car navigation problem. The neural controller is responsible for preventing the car from veering into canals found on either side of the road. Benchmark Lane Keeping models another safety-related car-driving problem. The neural controller aims to maintain a vehicle between lane markers and keep it centered in a possibly curved lane. The curvature of the road is considered as a disturbance input. Environment disturbances of such kind can be conservatively specified in our model, accounting for noise and unmodeled dynamics. Our verification approach supports these disturbances (verification condition (10)). Benchmarks n-Car platoon model multiple (n) vehicles forming a platoon, maintaining a safe relative distance among one another [39]. Each of these benchmarks exhibited some number of violations that were remediated by our verification methodology. Benchmark Oscillator consists of a two-dimensional switched oscillator plus a 16-order filter. The filter smoothens the input signals and has a single output signal. We verify that the output signal is below a safe threshold. Because of the model complexity of this benchmark, it exhibited significantly more violations than the others. Indeed, the neural-network controlled system often oscillated between the safe and unsafe boundary in many runs. Consequently, the overhead in this benchmark is high because a large number of shield interventions was required to ensure safety. In other words, the synthesized shield trades performance for safety to guarantee that the threshold boundary is never violated.

For all benchmarks, our tool successfully generated safe interpretable deterministic programs and inductive invariants as shields. When a neural controller takes an unsafe action, the synthesized shield correctly prevents this action from executing by providing an alternative provable safe action proposed by the verified deterministic program. In term of performance, Table 1 shows that a shielded neural policy is a feasible approach to drive a controlled system into a steady state. For each of the benchmarks studied, the programmatic policy is less performant than the shielded neural policy, sometimes by a factor of two or more (e.g., Cartpole, Self-Driving, and Oscillator). Our result demonstrates that executing a neural policy in tandem with a program distilled from it can retain performance, provided by the neural policy, while maintaining safety, provided by the verified program.

Although our synthesis algorithm does not guarantee convergence to the global minimum when applied to nonconvex RL problems, the results given in Table 1 indicate that our algorithm can often produce high-quality control programs, with respect to a provided sketch, that converge reasonably fast. In some of our benchmarks, however, the number of interventions are significantly higher than the number of neural controller failures, e.g., 8-Car platoon and Oscillator. However, the high number of interventions is not primarily because of non-optimality of the synthesized programmatic controller. Instead, inherent safety issues in the neural network models are the main culprit that triggers shield interventions. In 8-Car platoon, after corrections made by our deterministic program, the neural model again takes an unsafe action in the next execution step so that the deterministic program has to make another correction. It is only after applying a number of such shield interventions that the system navigates into a part of the state space that can be safely operated on by the neural model. For this benchmark, all system states where there occur shield interventions are indeed unsafe. We also examined the unsafe simulation runs made by executing the neural controller alone in Oscillator. Among the large number of shield interventions (as reflected in Table 1), 74% of them are effective and indeed prevent an unsafe neural decision. Ineffective interventions in Oscillator are due to the fact that, when optimizing equation (5), a large penalty is given to unsafe states, causing the synthesized programmatic policy to weigh safety more than proximity when there exist a large number of unsafe neural decisions.

### Suitable Sketches

Providing a suitable sketch may need domain knowledge. To help the user more easily tune the shape of a sketch, our approach provides algorithmic support by not requiring conditional statements in a program sketch and syntactic sugar, i.e., the user can simply provide an upper bound on the degree of an invariant sketch.

Our experimental results are collected using the invariant sketch defined in equation (7) and we chose an upper bound of 4 on the degree of all monomials included in the sketch. Recall that invariants may also be used as conditional predicates as part of a synthesized program. We adjust the invariant degree upper bound to evaluate its effect in our synthesis procedure. The results are given in Table 2.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Degree</th>
<th>Verification</th>
<th>Interventions</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pendulum</td>
<td>2</td>
<td>TO</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>22.6s</td>
<td>40542</td>
<td>7.82%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>23.6s</td>
<td>30787</td>
<td>8.79%</td>
</tr>
<tr>
<td>Self-Driving</td>
<td>2</td>
<td>TO</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24s</td>
<td>128851</td>
<td>6.97%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>25.1s</td>
<td>123671</td>
<td>26.85%</td>
</tr>
<tr>
<td>8-Car-platoon</td>
<td>2</td>
<td>172.9s</td>
<td>43952</td>
<td>8.36%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>540.2s</td>
<td>37990</td>
<td>9.19%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>TO</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Generally, high-degree invariants lead to fewer interventions because they tend to be more permissive than low-degree ones. However, high-degree invariants take more time to synthesize and verify. This is particularly true for high-dimension models such as 8-Car platoon. Moreover, although high-degree invariants tend to have fewer interventions, they have larger overhead. For example, using a shield of degree 8 in the Self-Driving benchmark caused an overhead of 26.85%. This is because high-degree polynomial computations are time-consuming. On the other hand, an insufficient degree upper bound may not be permissive enough to obtain a valid invariant. It is, therefore, essential to consider the tradeoff between overhead and permissiveness when choosing the degree of an invariant.

Handling Environment Changes. We consider the effectiveness of our tool when previously trained neural network controllers are deployed in environment contexts different from the environment used for training. Here we consider neural network controllers of larger size (two hidden layers with 1200 × 900 neurons) than in the above experiments. This is because we want to ensure that a neural policy is trained to be near optimal in the environment context used for training. These larger networks were in general more difficult to train, requiring at least 1500 seconds to converge.

Our results are summarized in Table 3. When the underlying environment slightly changes, learning a new safety shield takes substantially shorter time than training a new network. For Cartpole, we simulated the trained controller in a new environment by increasing the length of the pole by 0.15 meters. The neural controller failed 3 times in our 1000 episode simulation; the shield interfered with the network operation only 8 times to prevent these unsafe behaviors. The new shield was synthesized in 239s significantly faster than retraining a new neural network for the new environment. For (Inverted) Pendulum, we deployed the trained neural network in an environment in which the pendulum’s mass is increased by 0.3kg. The neural controller exhibits noticeably higher failure rates than in the previous experiment; we were able to synthesize a safety shield adapted to this new environment in 581 seconds that prevented these violations. The shield intervened with the operation of the network only 8.7 number of times per episode. Similar results were observed when we increased the pendulum’s length by 0.15m. For Self-driving, we additionally required the car to avoid an obstacle. The synthesized shield provided safe actions to ensure collision-free motion.

### Table 3. Experimental Results on Handling Environment Changes.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Environment Change</th>
<th>Neural Network Size</th>
<th>Neural Network Failures</th>
<th>Deterministic Program as Shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartpole</td>
<td>Increased Pole length by 0.15m</td>
<td>1200 × 900</td>
<td>3</td>
<td>1, 239s, 2.91%</td>
</tr>
<tr>
<td>Pendulum</td>
<td>Increased Pendulum mass by 0.3kg</td>
<td>1200 × 900</td>
<td>77</td>
<td>1, 581s, 8.11%</td>
</tr>
<tr>
<td>Pendulum</td>
<td>Increased Pendulum length by 0.15m</td>
<td>1200 × 900</td>
<td>76</td>
<td>1, 483s, 6.53%</td>
</tr>
<tr>
<td>Self-driving</td>
<td>Added an obstacle that must be avoided</td>
<td>1200 × 900</td>
<td>203</td>
<td>1, 392s, 8.42%</td>
</tr>
</tbody>
</table>

### 6 Related Work

#### Verification of Neural Networks.

While neural networks have historically found only limited application in safety- and security-related contexts, recent advances have defined suitable verification methodologies that are capable of providing stronger assurance guarantees. For example, Reluplex [23] is an SMT solver that supports linear real arithmetic with ReLU constraints and has been used to verify safety properties in a collision avoidance system. AF² [17, 31] is an abstract interpretation tool that can reason about robustness specifications for deep convolutional networks. Robustness verification is also considered in [8, 21]. Systematic testing techniques such as [43, 45, 48] are designed to automatically generate test cases to increase a coverage metric, i.e., explore different parts of neural network architecture by generating test inputs that maximize the number of activated neurons. These approaches ignore effects induced by the actual environment in which the network is deployed. They are typically ineffective for safety verification of a neural network controller over an infinite time horizon with complex system dynamics. Unlike these whitebox efforts that reason over the architecture of the network, our verification framework is fully blackbox, using a network only as an oracle to learn a simpler deterministic program. This gives us the ability to reason about safety entirely in terms of the synthesized program, using a shielding mechanism derived from the program to ensure that the neural policy can only explore safe regions.

#### Safe Reinforcement Learning.

The machine learning community has explored various techniques to develop safe reinforcement machine learning algorithms in contexts similar to ours, e.g., [32, 46]. In some cases, verification is used as a safety validation oracle [2, 11]. In contrast to these efforts, our inductive synthesis algorithm can be freely incorporated into any existing reinforcement learning framework and applied to any legacy machine learning model. More importantly, our design allows us to synthesize new shields from existing ones, without requiring retraining of the network, under reasonable changes to the environment.


An important reason underlying the success of program synthesis...
is that sketches of the desired program [41, 42], often provided along with example data, can be used to effectively restrict a search space and allow users to provide additional insight about the desired output [5]. Our sketch based inductive synthesis approach is a realization of this idea applicable to continuous and infinite state control systems central to many machine learning tasks. A high-level policy language grammar is used in our approach to constrain the shape of possible synthesized deterministic policy programs. Our program parameters in sketches are continuous, because they naturally fit continuous control problems. However, our synthesis algorithm (line 9 of Algorithm 2) can call a derivate-free random search method (which does not require the gradient or its approximative finite difference) for synthesizing functions whose domain is disconnected, (mixed-)integer, or non-smooth. Our synthesizer thus may be applied to other domains that are not continuous or differentiable like most modern program synthesis algorithms [41, 42].

In a similar vein, recent efforts on interpretable machine learning generate interpretable models such as program code [47] or decision trees [9] as output, after which traditional symbolic verification techniques can be leveraged to prove program properties. Our approach novelly ensures that only safe programs are synthesized via a CEGIS procedure and can provide safety guarantees on the original high-performing neural networks via invariant inference. A detailed comparison was provided on page 3.

**Controller Synthesis.** Random search [30] has long been utilized as an effective approach for controller synthesis. In [29], several extensions of random search have been proposed that substantially improve its performance when applied to reinforcement learning. However, [29] learns a single linear controller that we found to be insufficient to guarantee safety for our benchmarks in our experience. We often need to learn a program that involves a family of linear controllers conditioned on different input spaces to guarantee safety. The novelty of our approach is that it can automatically synthesize programs with conditional statements by need, which is critical to ensure safety.

In Sec. 5, linear sketches were used in our experiments for shield synthesis. LQR-Tree based approaches such as [44] can synthesize a controller consisting of a set of linear quadratic regulator (LQR [14]) controllers each applicable in a different portion of the state space. These approaches focus on stability while our approach primarily addresses safety. In our experiments, we observed that because LQR does not take safe/unsafe regions into consideration, synthesized LQR controllers can regularly violate safety constraints.

**Runtime Shielding.** Shield synthesis has been used in runtime enforcement for reactive systems [12, 25]. Safety shielding in reinforcement learning was first introduced in [4]. Because their verification approach is not symbolic, however, it can only work over finite discrete state and action systems. Since states in infinite state and continuous action systems are not enumerable, using these methods requires the end user to provide a finite abstraction of complex environment dynamics; such abstractions are typically too coarse to be useful (because they result in too much over-approximation), or otherwise have too many states to be analyzable [15]. Indeed, automated environment abstraction tools such as [16] often take hours even for simple 4-dimensional systems. Our approach embraces the nature of infinity in control systems by learning a symbolic shield from an inductive invariant for a synthesized program that includes an infinite number of environment states under which the program can be guaranteed to be safe. We therefore believe our framework provides a more promising verification pathway for machine learning in high-dimensional control systems. In general, these efforts focused on shielding of discrete and finite systems with no obvious generalization to effectively deal with continuous and infinite systems that our approach can monitor and protect. Shielding in continuous control systems was introduced in [3, 18]. These approaches are based on HJ reachability (Hamilton-Jacobi Reachability [7]). However, current formulations of HJ reachability are limited to systems with approximately five dimensions, making high-dimensional system verification intractable. Unlike their least restrictive control law framework that decouples safety and learning, our approach only considers state spaces that a learned controller attempts to explore and is therefore capable of synthesizing safety shields even in high dimensions.

## 7 Conclusions

This paper presents an inductive synthesis-based toolchain that can verify neural network control policies within an environment formalized as an infinite-state transition system. The key idea is a novel synthesis framework capable of synthesizing a deterministic policy program based on an user-given sketch that resembles a neural policy oracle and simultaneously satisfies a safety specification using a counterexample-guided synthesis loop. Verification soundness is achieved at runtime by using the learned deterministic program along with its learned inductive invariant as a shield to protect the neural policy, correcting the neural policy’s action only if the chosen action can cause violation of the inductive invariant. Experimental results demonstrate that our approach can be used to realize fully trustworthy reinforcement learning systems with low overhead.

## Acknowledgments

We thank our shepherd Swarat Chaudhuri and the anonymous reviewers for their comments and suggestions. This work was supported in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.
References


