RAVE: Enabling safety verification for realistic deep reinforcement learning systems

Wenbo Guo[2]*, **Taesung Lee**[3], **Kevin Eykholt**[1]*, **Jiyong Jang**[1]
[1]IBM Research, [2]Purdue University, [3]Bloomberg
henrygwb@purdue.edu, {kheykholt, jjang}@ibm.com, {elca4u}@gmail.com

Abstract

Recent advancements in reinforcement learning (RL) expedited its success across a wide range of decision-making problems. However, a lack of safety guarantees restricts its use in critical tasks. While recent work has proposed several verification techniques to provide such guarantees, they require that the state-transition function be known and the reinforcement learning policy be deterministic. Both of these properties may not be true in real environments, which significantly limits the use of existing verification techniques. In this work, we propose two approximation strategies that address the limitation of prior work allowing the safety verification of RL policies. We demonstrate that by augmenting state-of-the-art verification techniques with our proposed approximation strategies, we can guarantee the safety of non-deterministic RL policies operating in environments with unknown state-transition functions. We theoretically prove that our technique guarantees the safety of an RL policy at runtime. Our experiments on three representative RL tasks empirically verify the efficacy of our method in providing a safety guarantee to a target agent while maintaining its task execution performance.

1 Introduction

Reinforcement learning (RL) has shown great success for various sequential decision-making problems [25, 12]. However, ensuring that RL agents do not enter undesirable states is difficult due to the unpredictable nature of supervised policies, the use of non-determinism to encourage exploration and the long-term consequence of seemingly safe actions. This uncertainty raises serious safety and security concerns that limit their applicability in critical application fields.

Existing research has proposed various technical approaches to guarantee the safety of RL agents (*i.e.,* ensuring that an agent never enters into an unsafe state at runtime). These approaches can be generally categorized into: (a) training-time verification, which integrates the verification into the training process, thus guaranteeing a safe policy after training [1, 13, 3] and (b) post-training verification, which performs safety verification on a pre-trained RL policy and provides shielding for the target policy if the current policy is deemed unsafe [28, 31].

Although effective, these methods introduce rigorous assumptions/constraints to the agent and the corresponding environment, which significantly limits their applications in real deployments. Specifically, existing verification techniques require the state-transition function be known and the agent be deterministic; otherwise verification cannot be performed. These two assumptions, however, are often not supported in real-world applications. The state-transition function of most RL tasks are unknown, even in simulation environments such as MuJoCo and OpenAI GYM, and, instead, must be observed. For complex tasks, effective policies are sometimes non-deterministic [4].

*This work was done while the author was at IBM Research.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

We propose a novel safety verification method for non-deterministic RL policies executing in an environment with a possibly unknown state-transition function. Our method builds upon post-training verification methods. These methods create a program that approximates the agent's policy. The safety of the program is verified, and the program is shielded so that the agent is guaranteed to be safe with respect to a set of safety conditions. If executing an action from the original policy would violate the safety conditions, an action is instead drawn from the shielded program policy.

We extend this approach with a set of approximation strategies. If a non-deterministic agent is given, we use a method to transform the original policy into a categorical combination of two or more deterministic policies. We verify the safety of each deterministic policy and generate the respective shielding policy and safety conditions using existing solvers. With respect to environments with an unknown state-transition function, we propose approximating the unknown state-transition function with a neural network trained on observations of the state-action pairs. Once trained, we use a Taylor approximation to express the neural network as a polynomial function, which existing solvers can use for verification. At runtime, we check if the action produced by the non-deterministic agent violates the safety conditions of the underlying deterministic policy most likely to produce the action and use the corresponding shielding policy if so. We provide theoretical proof that RAVE guarantees the safety of the agent under such conditions. We also experiment with three common RL tasks demonstrating that our approach achieves similar performance as the original non-deterministic agent, but with a safety guarantee. We name our method as "**R**ealistic s**A**fety **VE**rification of RL Policies" (RAVE).

2 Key Techniques

2.1 Overview

RAVE is built upon prior post-training safety verification methods. In such works, the verification method V takes as input a polynomial state-transition function F and a deterministic policy π. The verification process outputs a shielding policy π_l and the safety condition (*i.e.,* inductive invariant ϕ) under which π_l should be executed instead of π. Prior works have demonstrated that augmenting π with π_l and ϕ guarantees safety in simple and oftentimes unrealistic scenarios. In our implementation, we build upon the post-training verification method of [31].

We must relax the requirement that the policy be deterministic and the system transitions be known as such conditions are unlikely to be satisfied in complex, real deployments. First, we propose *decomposing a non-deterministic policy* into a probabilistic combination of multiple deterministic policies. After decomposition, we can reuse prior verification methods to individually safeguard each deterministic policy with an associated shielding policy. At runtime, we use a *modified runtime shielding strategy* to guarantee the safe execution of a non-deterministic policy using the decomposed deterministic policies and their respective shielding policies. Next, we propose a *two-step approximation method* to enable safety verification in environments with unknown state-transition functions. First, we run the pre-trained neural policy π in the corresponding environment and collect a set of trajectories $\mathcal{T}$. These trajectories are used to fit a neural network N_θ, parameterized by θ, which takes as input the state and action at time step t and output the predicted state of the next time step $t+1$. This neural network can now be used as an approximation to the environment's unknown state-transition function F_π. As existing verification techniques work only for systems with polynomial state transitions, the neural network cannot be directly used for verification. Thus, we approximate the trained neural network N_θ with a polynomial function $\widetilde{N}$ through Taylor expansion [19]. This final polynomial approximation can be used with existing verification methods. We do not directly approximate $\mathcal{T}$ with polynomial functions because, compared to DNNs, polynomial functions have a limited capability in approximating complicated functions and handling high-dimensional inputs [20].

2.2 System Design of RAVE

Fig. 1 shows the overview of RAVE . Given a policy π with a set of user-defined initial states $\mathcal{S}_0$ and unsafe states $\mathcal{S}_u$, RAVE first identifies if policy decomposition or state-transition approximation is required. If so, it uses our proposed approach to generate a set of determinis-

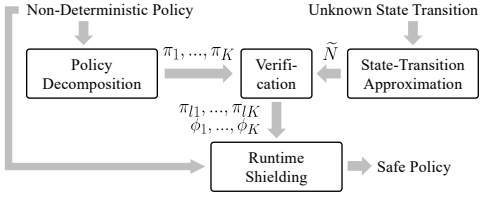

Figure 1: RAVE System Design.

tic policies $\pi_1, \ldots, \pi_K$ and a polynomial state-transition function. Then, using the verification strategy of [31], each deterministic policy π_k is processed and an alternative shielding policy π_{lk} and its corresponding safety condition/inductive invariant ϕ_k are generated. See Appendix B for more technical details.

Runtime Shielding. During runtime, at each time step t, RAVE generates an action using policy $\pi(s_t)$. Then, the K underlying deterministic policies the action was most likely to be sampled is identified using the following equation:

$$j = \text{argmin}_k \|\pi(s_t) - \pi_k(s_t)\|_2 \tag{1}$$

Finally, RAVE checks if the action satisfies ϕ_j, the inductive invariant of π_j. If ϕ_j holds, RAVE takes the action $\pi(s_t)$. Otherwise, it takes the action $\pi_{lj}(s_t)$.

Theorem 1 (Safety Guarantee of RAVE). *Following the above runtime shielding strategy, the policy π and the programmatic policies $\{\pi_{l1}, ..., \pi_{lK}\}$ in the environment guarantee that the agent never falls into any unsafe states, i.e., $\forall s_t$, $\phi_1(s_t) \vee ... \vee \phi_K(s_t)$ is True.*

Table 1: Performance results averaged across 500 trials using a Bernoulli policy. "NA" refers to the cases where almost all trials failed so step performance was not computed.

	Cartpole			Pendulum			Carplatoon		
Policy	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**
Baseline Policy 1	180.7	0.2	1	178.88	0.17	24	199	0.42	1
Baseline Policy 2	200.0	0.11	471	NA	0.002	500	16.79	0.037	4
Non-Deterministic Policy (P=0.75)	122.2	0.29	2	NA	0.022	499	182.14	0.31	1
RAVE Policy	121.1	0.54	**0**	185.96	0.67	**0**	183.92	0.62	**0**
Non-Deterministic Policy (P=0.5)	129.69	0.34	3	NA	0.009	500	46.41	0.08	2
RAVE Policy	129.43	0.6	**0**	181.97	0.63	**0**	42.95	0.14	**0**
Non-Deterministic Policy (P=0.25)	195.72	0.44	6	NA	0.006	500	19.89	0.03	5
RAVE Policy	197.74	6.13	**0**	155.72	0.59	**0**	19.48	0.06	**0**

Table 2: Performance results averaged across 500 trials using a Gaussian policy.

	Cartpole			Pendulum			Carplatoon		
Policy	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**
Non-Deterministic Policy	115.98	0.14	1	179.46	0.19	27	150.83	0.14	4
RAVE Policy (K=3, mode)	123.48	0.16	**0**	190.72	0.21	0	160.75	0.169	**0**
RAVE Policy (K=3, mean)	122.19	0.17	**0**	190.29	0.21	0	153.1	0.16	**0**
RAVE Policy (K=5, mean)	122.5	0.16	**0**	189.57	0.22	0	153.38	0.162	**0**
RAVE Policy (K=7, mean)	122.13	0.19	**0**	190.1	0.20	0	153.15	0.164	**0**

3 Evaluation

In this section, we evaluate RAVE on three widely used safety-critical environments: Cartpole [7], Pendulum [7], and Carplatoon [23]. For each of these environments, we evaluate RAVE using a non-deterministic policy both when the state-transition function is known and unknown.

Table 3: Performance results averaged across 500 trials using an unknown state-transition function and a Bernoulli policy.

	Cartpole			Pendulum			Carplatoon		
Policy	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**
Non-Deterministic Policy (P=0.75)	122.2	0.29	2	NA	0.022	499	182.14	0.31	1
RAVE Policy (P=0.75)	121.69	0.85	**0**	186.67	0.13	**0**	183.51	0.96	**0**
Non-Deterministic Policy (P=0.5)	129.69	0.34	3	NA	0.009	500	46.41	0.08	2
RAVE Policy	131.75	0.9	**0**	183.83	0.28	**0**	42.4	0.17	**0**
Non-Deterministic Policy (P=0.25)	195.72	0.44	6	NA	0.006	500	19.89	0.03	5
RAVE Policy	168.24	6.01	**0**	154.98	0.38	**0**	19.45	0.07	**0**

Table 4: Performance results averaged across 500 trials using an unknown state-transition function and a Gaussian policy.

	Cartpole			Pendulum			Carplatoon		
Policy	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**	**Avg. Steps**	**Avg. Runtime (s)**	**Failure Count**
Non-Deterministic Policy	115.98	0.14	1	179.46	0.19	27	150.83	0.14	4
RAVE Policy (K=3, mode)	115.39	0.14	**0**	187.72	0.2	0	160.4	0.23	**0**
RAVE Policy (K=3, mean)	114.17	0.17	**0**	187.72	0.2	0	160.54	0.19	**0**
RAVE Policy (K=5, mean)	115.33	0.14	**0**	187.88	0.24	0	160.46	0.223	**0**
RAVE Policy (K=7, mean)	114.69	0.14	**0**	178.5	0.19	0	160.75	0.177	**0**

3.1 Experiment 1: Non-Deterministic Policy with a Known State-Transition Function

Design. To test our runtime shielding method and our policy decomposition strategy, we construct two non-deterministic policies corresponding to the two types introduced in Section B: (a) a Bernoulli probabilistic mixture of multiple deterministic policies (N1), and (b) a Gaussian policy (N2).

For the Bernoulli non-deterministic policy, we first train two deterministic baseline policies. Then, we combine them with a certain probability p, *i.e.,* at each time step, the mixture policy takes an action given by the baseline policy 1 with the probability p and the baseline policy 2 with the probability $1 - p$. We vary $p = 0.25, 0.5, 0.75$ and evaluate the performance of the mixture non-deterministic policy with and without shielding using RAVE .

For the Gaussian non-deterministic policy, we first train a neural policy with a Gaussian distribution. Then, using our decomposition method, we split the policy into $K = 3, 5, 7$ deterministic policies. Finally, using each set of deterministic policies, we shield the original non-deterministic policy with RAVE . For $K = 3$, we report performance using both the mean and mode to represent each region.

Results. Table 2.2 shows the results of the Bernoulli policy. We observe that for most tasks, the baseline policy 1 is more secure, but requires more steps and has a longer runtime than the baseline policy 2. We also note that neither policy is completely safe. As such, the combined non-deterministic policies remain unsafe regardless of the value of p. However, with shielding, the RAVE policies all safely finish the scenario and report similar performance to the non-deterministic policies with respect to average steps. Regarding runtime, RAVE is about twice as slow as the non-deterministic policies in many cases. The runtime overhead of RAVE has two main contributing sources. First, RAVE constantly checks the safety condition ϕ, which adds a constant time overhead. Second, when the mixture policy takes an unsafe action, it will terminate prematurely as an unsafe state was entered. In contrast, RAVE prevents early terminations by switching to the inefficient, but safe shielding policy π_{lk}, thus extending episode length. The often unshielded mixture policies have lower runtime (*e.g.,* pendulum) due to unsafe behavior, whereas the average runtime for RAVE policy is high due constant shielding. This difference is most evident in the CartPole experiment when $p = 0.25$.

Table 2.2 shows the performance of the Gaussian policy. We observe that RAVE preserves the performance of the original policy while ensuring safety. These results confirm the effectiveness of RAVE 's runtime shielding and policy decomposition. Note that, the table also shows that RAVE is not sensitive to the choice of K and mean/mode. This is an important property as users do not need to exhaustively search for an ideal set of hyper-parameters to obtain a decent performance.

3.2 Experiment 2: Non-Deterministic Policy with an Unknown State-Transition Function

Design. We now measure the performance of RAVE when the state-transition function of each environment is unknown. For each non-deterministic policy used in Section 3.1, we collect a set of trajectories and apply our described state-transition approximation strategy to learn an approximated state-transition function for the task. Then, we use the approximated function during verification and compare the performance with and without shielding. In these experiments, we use a first-order Taylor approximation (*i.e.,* $I = 1$ in Eqn. (6)) .

Results. Before presenting the policy performance, we first describe the accuracy of our approximated state-transition functions. Specifically, we compute the average l_2-norm difference between the parameters in the approximated and actual transition functions. The results are as follows: CartPole-0.002, Pendulum-0.001, Carplatoon-0.02. These numbers suggest that our approximation strategy produces accurate approximations when the state-transition function is unknown. Table 3 and Table 4 show the verification results of the Bernoulli and Gaussian policies. As before, RAVE is able to ensure the safety of each non-deterministic policy and maintain the similar task performance with respect to average steps. There is a slightly higher runtime overhead. These results align with our theoretical analysis and validate RAVE ' effectiveness.

4 Conclusion

Reinforcement learning is widely used for sequential decision making problems, but the safety of RL agents needs to be guaranteed. Existing verification techniques are limited in scope as they can only verify the safety of deterministic agents operating in an environment with known state transitions.

This paper introduces RAVE , a safety verification technique for non-deterministic RL policies trained in environments with unknown state transitions. RAVE integrates two novel approximation strategies with the state-of-the-art verification technique. We demonstrate, both theoretically and empirically, that RAVE guarantees the runtime safety of a pre-trained agent while preserving the performance.

References

[1] Verma Abhinav, Yue Yisong Le Hoang, and Chaudhuri Swarat. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

[2] Kaspar Althoefer, Bart Krekelberg, Dirk Husmeier, and Lakmal Seneviratne. Reinforcement learning in a rule-based navigator for robotic manipulators. *Neurocomputing*, 2001.

[3] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In *Advances in Neural Information Processing Systems*, 2020.

[4] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[5] Osbert Bastani. Safe planning via model predictive shielding. *arXiv preprint arXiv:1905.10691*, 2019.

[6] Surya Bhupatiraju, Kumar Krishna Agrawal, and Rishabh Singh. Towards mixed optimization for reinforcement learning with program synthesis. In *International Conference on Machine Learning Workshop*, 2018.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Yash Chandak, Scott M Jordan, Georgios Theocharous, Martha White, and Philip S Thomas. Towards safe policy improvement for non-stationary mdps. In *Advances in Neural Information Processing Systems*, 2020.

[9] Christoph Dann, Lihong Li, Wei Wei, and Emma Brunskill. Policy certificates: Towards accountable reinforcement learning. In *International Conference on Machine Learning*, 2019.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[11] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. *arXiv preprint arXiv:1905.10615*, 2019.

[12] Shixiang Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. In *IEEE International Conference on Robotics and Automation*, 2017.

[13] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[14] Zhengyao Jiang and Shan Luo. Neural logic reinforcement learning. In *International Conference on Machine Learning*, 2019.

[15] Jens Kober and Jan Peters. Imitation and reinforcement learning. *IEEE Robotics & Automation Magazine*, 2010.

[16] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas A Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. *Nature Machine Intelligence*, 2020.

[17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[18] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[19] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 1976.

[20] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Neural Information Processing Systems*, 2017.

[21] Geir K Nilsen, Antonella Z Munthe-Kaas, Hans J Skaug, and Morten Brun. Efficient computation of hessian matrices in tensorflow. *arXiv preprint arXiv:1905.05559*, 2019.

[22] Pinar Ozisik and Philip S Thomas. Security analysis of safe and seldonian reinforcement learning algorithms. In *Advances in Neural Information Processing Systems*, 2020.

[23] Bastian Schürmann and Matthias Althoff. Optimal control of sets of solutions to formally guarantee constraints of disturbed linear systems. In *American Control Conference (ACC)*, 2017.

[24] David Silver. Lectures on reinforcement learning. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[25] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[26] Matteo Turchetta, Andrey Kolobov, Shital Shah, Andreas Krause, and Alekh Agarwal. Safe reinforcement learning via curriculum induction. In *Advances in Neural Information Processing Systems*, 2020.

[27] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI conference on artificial intelligence*, 2016.

[28] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 2018.

[29] C. Watkins. Learning from delayed rewards. *PhD thesis, University of Cambridge*, 1989.

[30] Wenbo Zhang, Osbert Bastani, and Vijay Kumar. Mamps: Safe multi-agent reinforcement learning via model predictive shielding. *arXiv preprint arXiv:1910.12639*, 2019.

[31] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

A Background

A.1 Deep Reinforcement Learning (DRL)

DRL concerns the sequential decision-making problems, where the environment is usually modeled as a Markov Decision Process (MDP) [17].

Definition 1 (Markov Decision Process). *A Markov Decision Process (MDP) is comprised of a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, F)$, in which $\mathcal{S}$ represents the state space, $\mathcal{A}$ denotes the action space, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $F : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ refers to the state-transition function.*

The goal of training a DRL agent is to learn a neural policy $\pi_\theta(a|s)$ that maximizes its expected total reward collected from a sequence of actions generated by that policy. Existing research has developed two main categories of training algorithms: deep Q-learning (*e.g.,* DDQN [27]) and policy-gradient methods (*e.g.,* DDPG [18]). Please refer to [24] for further technical details.

A.2 Safety Verification of a RL Policy

In addition to training an agent to maximize its expected rewards, the agent is also encouraged to avoid reaching certain undesirable states. While reward penalties can be used to discourage such actions, they do not guarantee an undesirable state will never be reached. Thus, many works have developed methods to provide formal safety guarantees with respect to an agent's behavior [1, 14, 13, 3, 28, 31, 6, 5, 30]. First, let us formally define **safety**:

Definition 2 (Safety of a DRL Policy). *Given a MDP with the state space $\mathcal{S}$, a set of user defined initial states $\mathcal{S}_0$, a set of user-defined unsafe states $\mathcal{S}_u$, if we can start from any state $s \in \mathcal{S}_0$ and never reach a state $s \in \mathcal{S}_u$ after following any possible sequence of actions generated from policy π, then we define policy π as safe.*

With this definition of safety, let us now focus on the state-of-the-art safety verification method by [31], which verifies the safety of a pre-trained deterministic RL policy with a known transition function. Their proposed method is broken into three steps: Program Learning, Program Verification, and Runtime Shielding. At a high level, they first perform a programmatic approximation of the neural policy. Then, they verify the safety of the program and generate an additional *shielding* policy along with a set of safety conditions. Finally, at runtime, the original neural policy is augmented with the shielding policy, which only executes if the neural policy's action would violate the safety conditions of the shielding policy. We focus on this particular work as `RAVE` uses the same underlying verification strategy, but proposes techniques to allow for the verification of non-deterministic RL policies with unknown state-transition functions.

Program Learning. Given a deterministic neural policy π, program learning synthesizes a linear program $\pi_l(s) ::= \text{return } \theta_l s$, to mimic π where $s \in \mathcal{S}$ is a state and θ_l is an imitation learning parameter [15]. θ_l is learned by solving the following objective function:

$$\theta_l = \text{argmax } d(\pi, \pi_l, \mathcal{T}) \,,$$
$$d(\pi, \pi_l, \mathcal{T}) = \sum_{T \in \mathcal{T}} \sum_{t \in T} \begin{cases} -\|\pi(s_t) - \pi_l(s_t)\|_2 & s_t \notin \mathcal{S}_u \\ -MAX & s_t \in \mathcal{S}_u \end{cases} , \tag{2}$$

where $\mathcal{T}$ is a set of trajectories collected by running π in the corresponding environment. This objective function penalizes action differences of π and π_l at the safe states.

Program Verification. By combining the linear program π_l defined above with the system's state-transition function F, we define a state-transition function F_{π_l}, such that $s_{t+1} = F_{\pi_l}(s_t) = F(s_t, \pi_l(s_t))$.

Program verification involves learning an inductive invariant ϕ, such that: 1) ϕ is disjoint with all unsafe states $\mathcal{S}_u$, 2) ϕ includes all initial states $\mathcal{S}_0$, and 3) all possible state transitions expressed by F_{π_l} is encapsulated in ϕ. Formally, $\phi \;::=\; E(s) \leq 0$ defines an inductive invariant where $E(s) : \mathbb{R}^n \to \mathbb{R}$ is a polynomial function that satisfies the following conditions:

$$\begin{aligned} &\forall s \in \mathcal{S}_u \;\; E(s) > 0 \,, \quad \forall s \in \mathcal{S}_0 \;\; E(s) \leq 0 \,, \\ &\forall (s, s') \in F_{\pi_l} \;\; E(s') - E(s) \leq 0 \,. \end{aligned} \tag{3}$$

As both F_{π_l} and E are polynomial functions, an SMT solver [10] can be used to find such an E that satisfies the conditions in Eqn. (3). In cases where the SMT solver cannot find a feasible solution,

[31] developed a Counterexample-guided Inductive Synthesis algorithm (See their work for more details).

Runtime Shielding. Runtime shielding uses the synthesized program π_l and learned inductive invariant ϕ to guarantee the safety of the neural policy at runtime. As the neural policy is likely to have better performance, at each time step t, we first generate an action from the neural policy $\pi(s_t)$ and predict the next state s_{t+1} resulting from the generated action. If the predicted next state s_{t+1} satisfies the inductive invariant (*i.e.,* $E(s_{t+1}) \leq 0$), then we take the action. Otherwise, we generate an action from the synthesized program π_l as its safety has been verified. In essence, π_l acts as a backup policy whenever the safety of the neural policy's action cannot be confirmed.

B More Technical details

First, we describe our **policy decomposition** approach that transforms a non-deterministic policy into a set of known deterministic policies. Next, we describe our **state-transition approximation** approach that produces a polynomial approximation of an unknown state-transition function using a series of observations. In both cases, our proposed approach results in transforming the system into a form that enables the reuse of existing verification techniques.

Policy Decomposition. Given a non-deterministic policy π, we consider two common structures:

- **N1:** π is a combination of K known deterministic policies. At runtime, first, one of the constituent deterministic policies $\pi_1, \ldots, \pi_K$ is selected according to a categorical distribution. Then, π returns the action generated by the selected deterministic policy.
- **N2:** π is a probability distributions in which values from the distribution can be directly mapped to an action in a continuous action space. At runtime, an action is drawn from the probability distribution.

N1 is often used as a mixture policy that combines different policy types, such as a neural policy and a rule-based policy, thus leveraging their complementary strengths [2]. As N1 is a combination of K known deterministic policies, we can directly decompose π in its constituent policies $\pi_1, \ldots, \pi_K$ and perform verification on each decomposed policy.

N2 draws an action directly from a probability distribution, such as a Gaussian distribution. For example, the rotation degree of the steering wheel can be randomly chosen from a Gaussian policy trained by the PPO algorithm [16]. To decompose N2, we propose dividing a continuous non-deterministic policy (*e.g.,* Gaussian or more generally exponential family distributions) into several regions and representing each region with a fixed sample. In doing so, the policy is approximated as probabilistic combination of a set of fixed samples, which can be though of as a combination of deterministic policies (*i.e.,* an N1 policy structure) and be verified as such. In the following, we use the Gaussian distribution to illustrate our technical approach. Given a Gaussian policy π with mean μ and variance σ^2, we divide its probability density function into K bins of equal area. Thus, if we draw a sample from π, it will be equally likely to belong to any of the bins. Then, we select a sample from each bin to represent the bin's region, denoted as $\mu_1, ..., \mu_K$. With these representative samples, the original Gaussian distribution can be approximated as a categorical distribution across the representative samples $\mu_1, ..., \mu_K$ with a equal probability $\frac{1}{K}$ to be selected. As part of the decomposition process, we must ensure that the mean and variance of the categorical distribution matches with that of the original policy π, i.e., the following equations hold:

$$\sum_k \mu_k \frac{1}{K} = \mu \,, \quad \sum_k (\mu_k - \mu)^2 \frac{1}{K} = \sigma^2 \,. \tag{4}$$

To make sure μ_k reflects the statistical significance of the $k-$th bin, we use either the mean or the mode of the first $K-1$ bins as $\mu_1, ..., \mu_{K-1}$, and compute μ_K based on Eqn. (4).

Note that, in the real-world, there is another type of non-deterministic policies that use N2-type policies as the part of the constituent policies in N1 to encourage both exploratory and exploitation (*e.g.,* the ϵ-greedy approach [29]). Our method can also be used with such cases as policy decomposition can be recursively performed for each N2-type sub-policy.

State-transition Approximation. Given an unknown state-transition function, we perform a two-step approximation process to obtain a polynomial state-transition function. The first step is to train a

neural network N_θ to mimic the unknown state-transition function. We collect a dataset of M trajectories $\mathcal{T}$ by running the original policy π in the environment. A trajectory $T \in \mathcal{T}$ consists of a sequence of the state and selected action at each time step, i.e., $\{s_0, a_0, s_1, a_1, ..., s_{|T|-1}, a_{|T|-1}, s_{|T|}\}$. The neural network is trained on $\mathcal{T}$ using the following objective function:

$$\text{argmin}_\theta \frac{1}{M} \sum_{T \in \mathcal{T}} \frac{1}{|T|} \sum_{t \in T} \|s_{t+1} - N_\theta([s_t, a_t])\|_2^2 , \tag{5}$$

where $[s_t, a_t]$ is the concatenation of s_t and a_t. Eqn. (5) updates θ to minimize the mean squared error between the true next state s_{t+1} and the prediction given by N_θ. Once trained, the second step of the approximation process is to transform the neural network N_θ into a polynomial function $\tilde{N}$ using a I-th order Taylor approximation:

$$\tilde{N}(\mathbf{x}) = \sum_{i=0}^{I} \frac{N_\theta^{(i)}([s_t, a_t])}{i!} (\mathbf{x} - [s_t, a_t])^i , \tag{6}$$

We use the Taylor approximation because of its flexibility. By varying I, we can increase/decrease approximation accuracy at the cost of complexity. As $\tilde{N}$ is a polynomial function, it can be used during verification to approximate the unknown state-transition function.

Proof of Theorem 1 At any initial state $s_0 \in \mathcal{S}_0$, we know $\phi_1(s_0) \wedge ... \wedge \phi_K(s_0)$ is True by definition. Now, at some arbitrary time t, let us assume $\phi_1(s_t) \vee ... \vee \phi_K(s_t)$ is True. Since $\phi_1(s_t) \vee ... \vee \phi_K(s_t)$ is True, we know that for some value k, $\phi_k(s_t)$ is also true. According to Eqn. (3), we have $E_k(s_{t+1}) - E_k(s_t) \leq 0$ and as we know $E_k(s_t) \leq 0$, it must also be True that $E_k(s_{t+1}) \leq 0$. This means that $\phi_1(s_{t+1})$ is True. Since $\phi_1(s_{t+1}) \vee ... \vee \phi_K(s_{t+1})$ holds for $t+1$, we can conclude that $\forall t \in \mathbb{R}^+, \phi_1(s_t) \vee ... \vee \phi_K(s_t)$ is True. □

C Experimental Setup

CartPole. This environment has a pole standing on top of a cart. The RL agent can move the cart horizontally along a frictionless track to keep the pole upright. The system is in an unsafe state if the angle of the pole is more than 30 degrees from being vertical or the cart moves more than 0.3 meters from the origin.

Pendulum. This environment contains a pendulum, starting at a random position and rotating around the circle center. The RL agent swings the pendulum to keep the pendulum upright. The system is in an unsafe state if the pendulum's angle is more than 23 degree from being upright.

Carplatoon. This environment models a real-world scenario where 4 cars form a platoon on a road and drive along the same direction. The RL agent can modify the horizontal and vertical speed of each car The system is in an unsafe state if the relative distance between two cars is less than a certain number [23].

In the above three environments, an episode directly terminate if it enters any unsafe state; otherwise, it will terminate when the agent's reward reaches a pre-defined value or if the max episode length is reached.

Evaluation Metrics. We evaluate `RAVE` on three aspects – task performance, efficiency, and safety. Task performance is measured by the average number of steps per episode required to complete the task, *i.e.,* the number of steps needed for a good termination (Avg. Steps). A smaller number means the agent could finish the task quicker, indicating better performance. Efficiency is measured by the average run time before termination of each episode (Avg. Runtime (s)). The safety is measured by the number of episodes in which the agent entered an unsafe state (Failure Count).

Implementation and Configuration. We implement `RAVE` in Python by using the code package of [31]. All experiments were conducted on a MacBook Pro laptop with 8 CPU cores. The laptop was not equipped with any GPU.

D Discussion

Computational Efficiency. The state-transition approximation process introduces additional computational overhead during verification as it involves training a DNN and conducting a Taylor approximation. If the environment involves a higher-order approximation such as a second-order approximation, which requires computing the Hessian matrix, the overhead costs are likely to significantly increase. In such cases, we could consider leveraging more efficient computation methods to reduce the runtime overhead of RAVE (*e.g.,* [21] to compute the Hessian matrix).

Generalization to Other Environments. In Section 2, we noted that our method can be applied to MDPs (single-player DRL tasks) with arbitrary unknown state transitions. The key challenge is how to effectively compute higher-order derivatives for inputs to a neural network. As part of future work, we will investigate if there are methods in the optimization/numerical computing community that can be used to enable a higher-order polynomial approximation. Going beyond single-player tasks, we will also explore how to generalize our method to multi-player environments. We plan to follow the idea of a recent work [11] and explore whether we can transform an original multi-player environment into a single-player environment for the target player (agent) and then apply our method to safeguard that agent in the transformed environment.

Limitations and Future Works. First, our current state-transition approximation gives one approximated function for a given environment, but it could be finer-grained. Specifically, we could divide the initial states into subsets and conduct an approximation for each subset. This would enable a more accurate approximation of the original state-transition function and generalize our method to more complicated systems. In the future, we will explore how to conduct such a piece-wise approximation effectively and extend our runtime shielding strategy to utilize the programs obtained from each approximated state-transition function. Second, our method cannot handle non-deterministic policies with a categorical distribution due to the limitations of existing constraint solvers. As future work, we plan to design a policy decomposition strategy for such policies so that the decomposed policies can be used with existing constraint solvers. Finally, we designed RAVE as a way to expand the application areas of post-training verification techniques. We are exploring how to adapt our approach to also improve training-phase verification techniques.

E Related Works

There are two main setups to guarantee the safety of DRL policies – learn a verified safe policy during training or perform post-training safety verification and enforcement.

Training Phase Safety Verification. These works incorporate safety as part of the training process. One method uses a two step approach. First, a programmatic policy is learned. Then, its safety is verified using program verification (automated reasoning) techniques (*e.g.,* constrained solvers) during training [1, 14, 13, 3]. For example, Abhinav *et al.* first trains an oracle DNN policy [1]. To enable safety verification with existing techniques, they then transform the DNN policy into a programmatic policy using imitation learning and output this programmatic policy as the final policy. Later works improved the performance of this approach by using more advanced learning methods [14, 13, 3]. However, as this approach relies on existing program verification tools (constrained solvers), its use has been restricted to a small set of environments.

An alternative method for training phase safety verification is to embed the safety guarantee as part of the optimization objective. Specifically, this method models the safety condition as additional constraints for the policy training objective function (*i.e.,* maximizing Q or V function). In doing so, safe policies can be learned by simply solving this modified constrained optimization problem (*e.g.,* via Seldonian algorithm [8]). While novel, this method can only be used in environments where the safety conditions can be formulated as optimization constraints. Compared to these works, our proposed approach seeks to guarantee the safety of a pre-trained policy, rather than incorporate safety as part of the training process.

Post-training Safety Verification. These works verify and enforce the safety of a pre-trained RL policies [28, 31, 6, 5, 30]. First, the pre-trained policy is approximated with a programmatic policy and then the programmatic policy is verified. One example is the work by Zhu *et al.* [31]. Their work can be broken down into three steps: approximation, verification, and runtime shielding. First, they use imitation learning to mimic the outputs of the pre-trained policy network. Then, they model

the safety conditions as inductive invariants and input the conditions along with the approximated policy into a constrained solver (*e.g.,* Z3 [10]). The solver outputs a set of environment constraints representing the states in which the approximated policy is guaranteed to be safe. In other words, if the current state is within the solver's output constraints, then actions originating from the programmatic approximation will never result in reaching an unsafe state. Thus, the programmatic approximation becomes a shielding policy. At runtime, actions are obtained first from the pre-trained policy and are checked for safety. If the action results in a state outside of states supported by the shielding policy, the system instead obtains an action from the shielding policy due to its safety guarantees. Otherwise, the action from the pre-trained policy is taken as it is more efficient.

Existing post-training verification techniques require the state-transition function of the environment to be known as this is a required input for the solvers. The solvers also require that the environment be linear or polynomial and that the pre-trained policy be deterministic. If any of the above are false, then existing solvers are unable to verify the programmatic approximation and generate the shielding policy. Our work builds upon an existing verification technique by proposing a set of strategies that allow us to transform unknown state-transition functions and non-deterministic policies into a form that can be utilized by existing solvers.

Other Safety Verification-related Techniques. In addition to the two main approaches discussed above, there are some other related methods for ensuring the safety of RL algorithms. For example, recent research [26, 22, 8] designed techniques to improve the efficacy, efficiency, and stability of a typical safe learning method – Seldonian algorithm. Some other research efforts have been made to provide safety guarantees for non-DRL tasks (*e.g.,* Tabular MDPs [9]) or explore the transferability of safety guarantee. Due to the task and setup differences, we do not consider these works in this paper.