Robustness to Adversarial Attacks in Learning-Enabled Controllers

No Author Given

No Institute Given

Abstract. Learning-enabled controllers used in cyber-physical systems (CPS) are known to be susceptible to adversarial attacks. Such attacks manifest as perturbations to the states generated by the controller's environment in response to its actions. We consider state perturbations that encompass a wide variety of adversarial attacks and describe an attack scheme for discovering adversarial states. To be useful, these attacks need to be natural, yielding states in which the controller can be reasonably expected to generate a meaningful response. We consider shield-based defenses as a means to improve controller robustness in the face of such perturbations. Our defense strategy allows us to treat the controller and environment as black-boxes with unknown dynamics. We provide a two-stage approach to construct this defense and show its effectiveness through a range of experiments on realistic continuous control domains such as the navigation control-loop of an F16 aircraft and the motion control system of humanoid robots.

Keywords: Robust Control · Deep Reinforcement Learning · Adversarial Defense.

1 Introduction

Deep reinforcement learning (RL) approaches have shown promise in synthesizing high-quality controllers for sophisticated cyber-physical domains such as autonomous vehicles and robotics [3,6]. However, because these domains are characterized by complex environments with large feature spaces, they have proven vulnerable to various kinds of adversarial attacks [9] that trigger violations of safety conditions the controller must respect. For example, a safe controller for an unmanned aerial navigation system must account for a myriad of factors with respect to weather, topography, obstacles, etc., that may maliciously affect safe operation, only a fraction of which are likely to have been considered during training. Ensuring that controllers are robust in the face of adversarial attacks is therefore an important ongoing challenge.

Although the state space over which the controller's actions take place is intractably large, the need to conform to physical realism greatly reduces the attack surface available to an adversary in practice. For example, an adversarial attack on a UAV cannot generate an obstacle from thin air, suspend gravity, or instantaneously eliminate prevailing wind conditions. A meaningful attack is

thus expected to generate states that are natural [9], i.e., states that can be derived from perturbations of realizable states as defined by the physics of the application domain under consideration. We are interested in identifying such states along a trajectory representing a rollout of the controller's learnt policy.

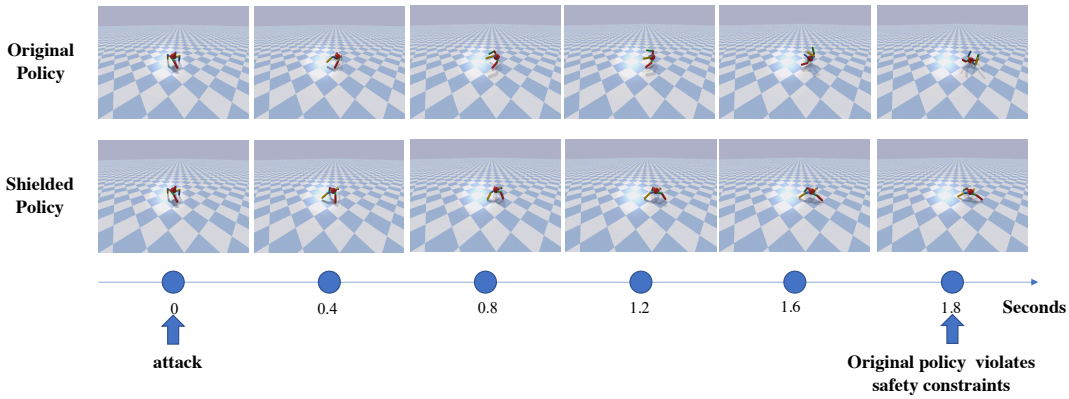

Fig. 1: Simulation of a shielding defense against an adversarial attack on an Ant robotics controller. The top row shows the effect of a successful attack. The bottom row illustrates an execution starting from the same initial (unsafe) state augmented with a shielded defense.

Successful attacks manifest as a safety failure in this policy. While these kinds of falsification methods are certainly important [7, 8] to assess safety specifications on a controller, devising a defensive strategy that prevents unsafe operation resulting from such attacks is equally valuable. In black-box settings where a policy's internal structure is unknown, shield-based defenses can be applied [1, 13]. One illustrative example is provided in Fig 1. An unnoticeable natural perturbation to the ant robot is applied at 0 seconds. For the policy without the defenses with shield, this ant robot flipped and thus violates the safety constraints. While with the defense provided by the shield, the ant moves as normal.

This paper considers the generation of such defenses that comprise three distinct components: (1) a mechanism to efficiently explore adversarial attacks within a realistic bounded area of the states found in simulated and seemingly safe trajectories; (2) a detector policy that predicts safe and unsafe states based on previously observed successful attacks; and, (3) an auxiliary policy that is triggered whenever the detector identifies an unsafe state; the role of the policy is to propose a new action that re-establishes a safe trajectory. To be practical, each of these components requires new insights on adversarial attack and defense in RL. While an effective defense must be tuned with respect to the detector to protect against identified attacks, any useful methodology must also be able

to generalize the shield to successfully prevent unseen attacks, i.e., attacks for which the detector has not been trained against.

Our contributions are 3-fold:

1. We present a new adversarial testing framework that identifies natural unsafe states found near safe ones in a simulated trajectory of learning-enabled controllers. The RL controller is guaranteed to produce trajectories that lead to safety violations from these states.
2. We learn detector classifiers from these unsafe trajectories that classify safe and unsafe states, and auxiliary policies triggered when the detector identifies an unsafe state; applying a shield on a presumed unsafe state yields a new safe trajectory, i.e., a trajectory that satisfies desired safety properties.
3. We provide a detailed experimental evaluation over a range of challenging RL benchmarks, including the PyBullet suite of robotic environments and the F16 ground collision avoidance system. Our results justify our claim that the robustness of black-box RL-enabled controllers can be significantly enhanced using learning-based detection and shielding approaches, without noticeable loss in performance.

2 Background

2.1 Discrete-time Continuous Systems

We consider discrete-time systems $M = \langle S, A, f, R\rangle$ with non-linear dynamics f, continuous state space $S \subseteq \mathbb{R}^n$ and action space $A \subseteq \mathbb{R}^k$. Here $s_{t+1} = f(s_t, u_t)$ where $s_t \in S$ is the state at time step t and f is a complex unknown nonlinear function. The objective, given a reward function $R : S \times A \to \mathbb{R}$, is to find a policy $\pi : S \to \Delta(A)$ where $\Delta(A)$ is a probability distribution over A that maximizes the discounted return $\mathbb{E}_\pi[\sum_{t=0..T} \gamma^t R(s_t, a_t)]$ where $\gamma \in (0, 1]$ is the discount factor. Given a policy π, a trajectory $\tau = (s_0, ..., s_T)$ is the sequence of states obtained when $s_{t+1} = f(s_t, \pi(s_t))$ up to a horizon T. Following deterministic control setting [21], the f and π are deterministic after deployment (i.e., the π always return the mean of $\Delta(A)$ as action), but in the learning phase, π can be stochastic.

2.2 Bayesian Optimization

Bayesian optimization (BO) is a black-box optimization approach that aims to find the global minimum for a black-box function $F(x)$ by querying the function a constrained number of times. More specifically, BO models $F(x)$ with a prior (frequently Gaussian Process-based) and uses an acquisition function to query values of $F(x)$ while trading off exploration and exploitation. BO is suitable for the case where evaluating the objective function often requires significant computational resources. It is an important motivation for us to choose the BO as the attack technique. Black-box attack approach requires a sequence of queries to the system under attack, but evaluating the cumulative reward over trajectory for one query can be computationally expensive.

3 Problem Statement

Consider a continuous-state system M where user-defined *unsafe* states (denoted as S_u) and *safe* states form a partition of the entire state space S of M.

A rollout of a trained agent policy π_o on M yields a state trajectory $\tau = (s_0, ..., s_T)$. Let $\mathcal{T}(\pi_o, M, S_o)$ be the set of *safe* trajectories formed by rollouts of policy π_o in the system M with initial state space S_o (*i.e.* trajectories in this set do not contain any states in S_u).

Define a ε-state perturbation of a state s to be another state $s' \in S$ that is within ε distance away from s by varying dimensions in s. We choose the L_∞ norm to measure state distances. Given a state s, denote $P_\varepsilon(s)$ to be the set of all possible perturbations on s.

We define an *Adversarial State Attack* $\texttt{Att}_\varepsilon(\tau)$ to be a set of states in which each state $s \in \texttt{Att}_\varepsilon(\tau)$ is an ε-state perturbation on a state in the trajectory τ that causes the new trajectory to be unsafe. In other words, $s \in \texttt{Att}_\varepsilon(\tau)$ if, for some k and s', $0 \leq k \leq T$, $s' \in P_\varepsilon(s_k)$, and executing the policy π_o from s' leads to a state in S_u. This can be thought of as a state perturbation at an arbitrary time index along the trajectory that results in safety failure. These perturbations can be minute, a small shift of a robot due to a bump on the ground, or mild turbulence affecting the positioning of a jet in midair. If ε is unconstrained, perturbations can be unrealistic, unlikely to occur in practice, or impossible to defend against given existing environment transition dynamics. We thus concern ourselves with values of ε that lead to perturbed states in Sec. 5.1.

We aim to find a way to defend against adversarial state attacks leading to a more robust control strategy that is still performant. In real-world scenarios, it can be hard to have access to both controller internals (underlying control policy) and environment conditions. Thus, our defense approach treats the original policy π_o as a black box and make no assumptions about the (hidden) environment dynamics. With these constraints in mind, we divide this problem into two stages. The first step is to detect adversarial state attacks, knowledge of which can be used to act differently, by shielding π_o from malicious state perturbations (the second step). One defense outline is in the style of [1] where a shield modulates actions taken by π_o based on the observation provided by the environment. We construct a two-level structure to represent the shield shown in Fig 2. The upper level would be able to detect that an $\texttt{Att}_\varepsilon$ attack has occurred. We call it the *detector*. While the lower level would be a switch mechanism between two policies. The original policy π_o is trained with the performance objective, which is shipped with the training task itself. The auxiliary policy π_{aux} focuses on the safety aspect. However, the safety objective is not directly provided by the training task itself, but the task defined the safety specifications. We translated the safety specifications of different tasks to safety objective and used it to train the π_{aux}.

With this setup, the constructed defense then uses the detector to decide whether to follow the original policy π_o or switch to π_{aux}. If these two functions are accurate, this approach is robust to adversarial state attacks for a given ε.

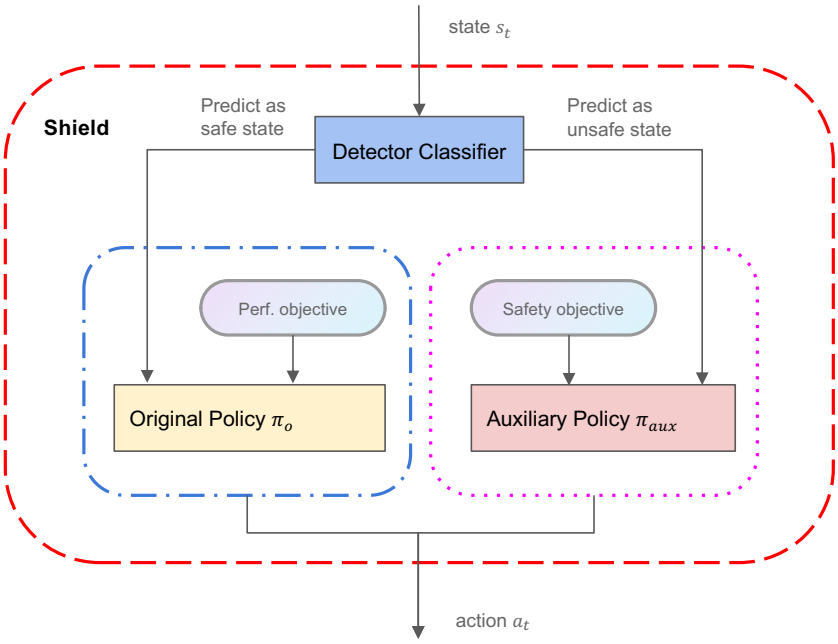

Fig. 2: Shield-Based Defense

Our goal is thus to learn these two functions keeping in mind the restrictions placed on the defense scheme.

4 Approach

Based on the definition of adversarial state attack $\texttt{Att}_\varepsilon$ (Sec. 3), given a trained policy π_o, a system M with some initial states S_o, we aim to find a set of state-based adversarial attacks $\texttt{Adv}_\varepsilon(\pi_o, M, S_o)$ defined as:

$$\texttt{Adv}_\varepsilon(\pi_o, M, S_o) = \{s \mid s \in \texttt{Att}_\varepsilon(\tau) \wedge \tau \in \mathcal{T}(\pi_o, M, S_o)\}$$

Our defense strategy follows the outline described in Fig 2.

4.1 Safety Specifications

We consider a safety specification φ consisting of multiple predicates denoted as ρ connected using Boolean operators including conjunction and disjunction.

$$\varphi := \rho \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$
$$\rho := t = t' \mid t \neq t' \mid t \leq t' \mid t < t' \mid t \geq t' \mid t > t'$$

A term t is any real-valued function defined over system variables. For example, consider a robot with the height of the center of mass being z. One safe height specification φ_{safe} is $z_{safe} < z$ where z_{safe} is a height threshold.

4.2 Objectives

We define the safety objective first. The safety objective is necessary for a training procedure of π_{aux}. In addition, the search problem for $\texttt{Adv}_\varepsilon(\pi_o, M, S_o)$ also requires a precise objective that is easy for optimization. We define the safety objective as $L(\varphi)$. A key feature of $L(\varphi)$ is that, given a state $s \in S$, φ holds on s iff $L(\varphi)(s) > 0$. Suppose that φ_1 requires that the height of the robot z should be higher than z_{safe}, and φ_2 requires that the velocity of the robot should be smaller than v_{safe}. In this case, we let $L(\varphi_1 \wedge \varphi_2)(s) = min(L(\varphi_1)(s), L(\varphi_2)(s)) = min(z - z_{safe}, v_{safe} - v)$, where z and v are two dimensions of s. To support all the definitions in Sec. 4.1, we define $L(\varphi) : S \to \mathbb{R}$ recursively.

$$
\begin{aligned}
L(t < t') &:= t' - t \\
L(t \neq t') &:= L(t < t' \vee t > t') \\
L(t = t') &:= [t = t'] \\
L(t \leq t') &:= L(t < t' \vee t = t') \\
L(\varphi \wedge \varphi') &:= \min(L(\varphi), L(\varphi')) \\
L(\varphi \vee \varphi') &:= \max(L(\varphi), L(\varphi'))
\end{aligned}
$$

where $[\cdot]$ is an indicator function. It returns 1 iff the Boolean formula in [ ] is true. Otherwise, it returns 0. Since $<$ and $>$, $\leq$ and $\geq$ are symmetries, one can change the positions of t and t' above if needed. For a trajectory τ, its *safety reward* is

$$L(\varphi)(\tau) = \min_{s \in \tau} L(\varphi)(s)$$

The performance objective is shipped with different tasks. Usually this is defined based on a robot's internal structure and the goal it wants to achieve, e.g., high moving velocity and low electricity cost. The design of the $L(\varphi)$ and performance objective unavoidably requires the knowledge of properties for tasks, e.g., the weight of a robot, the max speed of a joint. This is required by most RL tasks, especially in the continuous control domain. This knowledge is typically readily available before people define one specific learning tasks[1].

4.3 Attack

With $L(\varphi)$ defined above as an effective proxy for safety measurement, given a (set of) trajectory τ of a system M from S_o i.e. $\tau \in \mathcal{T}(\pi_o, M, S_o)$, we search in $\bigcup_{s \in \tau} P_\varepsilon(s)$ for a set of state S' with the objective of minimizing the safety reward

[1] See Appendix B. for details.

$L(\varphi)(\tau')$ where $\tau' \in \mathcal{T}(\pi_o, M, S')$. Observe that $L(\varphi)(\tau')$ is essentially a function over states (parameterized by the initial state of τ' and the policy π_o). The ability of Bayesian Optimization (BO) in optimizing black-box functions (e.g. $L(\varphi)(\tau')$) makes it qualified as an optimization scheme for our purpose. We use BO to find states in $\texttt{Adv}_\varepsilon(\pi_o, M, S_o)$ that lead to trajectories with a negative $L(\varphi)$ reward and are thus unsafe. In our experiments, we use EI (Expected Improvement) [12] as the acquisition function and GP (Gaussian Processes) [18] as the surrogate model.

4.4 Defense

We present the details of our defense approach in this section and its theoretical analysis in Appendix A.

Training the Detector. Our defense approach must be aware of potential unsafe states that occur when using the original policy π_o. To this end, we train a classifier (the detector) with data derived during the attack phase. When we attack policy π_o with BO, we obtain batches of trajectories via simulation in the environment, some of which starting from states in $\texttt{Adv}_\varepsilon(\pi_o, M, S_o)$ are indeed unsafe and the rest are still safe. For a safe trajectory, we label all included states as safe, while for an unsafe trajectory τ, all states that follow a state found in $\texttt{Att}_\varepsilon(\tau)$ are labelled as unsafe. Training a classifier with this data yields a detector that differentiates between states that are safe from those that produce unsafe trajectories affected by states in $\texttt{Adv}_\varepsilon(\pi_o, M, S_o)$.

Auxiliary Policy. When running a system, we use the detector to monitor current system states. If the detector identifies a potentially vulnerable state, the defense strategy must yield an alternative action to prevent the agent from entering an unsafe region. We use an auxiliary policy π_{aux} to provide such an action. To obtain the final building block of our defense, we use the trained detector to provide a reward signal for training π_{aux} using reinforcement learning. Specifically, we train π_{aux} with a reward function R_{safe} that has two components. First, the safety reward defined in Sec. 4.3 is considered part of R_{safe}. We also integrate the detector as a regularizer R_d for R_{safe}. For an input state s of the reward function, if the detector finds s safe, $R_d(s)$ gives a reward 1 and otherwise a reward 0 is given. Given a trajectory τ, define $R_d(\tau) = \sum_{s \in \tau} R_d(s)$ where c is a constant and $c \geq 0$. A high value of $R_d(\tau)$ means that many states in τ are deemed safe by the detector and is preferred. We use PPO [19] and DDPG [20] to train π_{aux} with the reward function R_{safe}. Formally,

$$R_{safe}(\tau) = L(\varphi)(\tau) + c \cdot R_d(\tau)$$

Adaptive Attack After we deploy the shield, it is still possible to apply the same attack procedure to adaptive attack our shield. On the other hand, because

our defense introduced a neural network detector, which can also be subject to an adaptive attack that exploits the model internals and deceives the detector using well-crafted and potentially risky states. Given a known unsafe system trajectory τ that can be defended by our detector (and the auxiliary policy), an attack on the detector seeks to identify states in τ that are vulnerable. Given a random attack function $\texttt{select}(\tau, \mathcal{F})$ that draws states from τ with frequency $\mathcal{F}$, for every state $s \in \texttt{select}(\tau, \mathcal{F})$, we search an adversarial state $\bar{s}$ such that:

$$\arg\min_{\bar{s} \in P_\varepsilon(s)} (p_{unsafe}(\bar{s}) - p_{safe}(\bar{s}))$$

Here, $p_{unsafe}(\bar{s})$ and $p_{safe}(\bar{s})$ represent the unsafe and safe probability for a state $\bar{s}$ predicted by the detector. The idea of such an adversarial attack is to apply perturbation to a state s to avoid detection [11] by minimizing the difference between the probabilities of labeling $\bar{s}$ as unsafe and safe by the detector. We empirically show in Sec. 5 that crafting attacks on the detector is unlikely to bypass our defense mechanism even using an unnatural high attack frequency $\mathcal{F}$ in settings where we can access the gradient of the detector.

5 Experiments

5.1 Selecting an Attack Range

As mentioned in Sec. 3, an adversarial state attack $\texttt{Att}_\varepsilon$ requires a meaningful ε-state perturbation. We choose ε with the following strategy. For a specific robot in the PyBullet benchmarks, we initialized our ε to be 0.001. We randomly perturbed initial states and run simulations 1000 times for every available policy. If there is a policy that does not find any unsafe states, we increase ε by 0.0005, and repeat this process until unsafe states are found in all policies. For all the other benchmarks, constraints on these ranges were available and used directly.

5.2 Attack Results

Table 1 shows two attack strategies. The first, rand. attack indicates the percentage of safety violations detected when initializing the state randomly around a sampled trajectory with respect to the chosen ε. For each benchmark, we randomly sampled 40K states within the bounds determined by ε and initialized the simulations accordingly. Column BO attack shows the attack success rate when using BO to discover attacks. The GPs of the method are initialized with 10 randomly sampled states and the BO algorithm samples the safety reward 30 times. For each state in one trajectory, we generate 40 sampled trajectories. If the length of an attacked trajectory was L, $40 \times L$ trajectories would be collected in one attack. We then count the unsafe trajectories in these simulations.

Not all features in an application are equally related to safety specifications. Taking the Humanoid robot as an example - its safety specification φ is to stay upright ($z > 0.78$). This goal is directly dependent on the height of the robot;

Table 1: Experimental Results. Benchmarks are of the form, E-A where E is the Environment and A is the type of controller attacked. The details of these benchmarks are provided in the Appendix B

Benchmarks	state/action dim	simu. traj. length	Attack ε	rand. attack	BO attack	defense succ. rate	attack improvement using shielded policy
Hopper-a2c	15/12	1000	0.001	1.29%	2.18%	91.40%	43.51%
Hopper-ppo				0.41%	5.27%	97.80%	27.97%
Hopper-trpo				1.71%	1.97%	97.90%	48.16%
HalfCheetah-a2c	26/17	1000	0.02	0.36%	9.43%	91.40%	34.46%
HalfCheetah-acktr				0.90%	14.54%	92.80%	65.23%
HalfCheetah-ddpg				1.63%	11.84%	88.20%	67.27%
HalfCheetah-ppo				0.38%	4.68%	97.80%	50.53%
HalfCheetah-sac				4.40%	5.88%	97.90%	62.12%
HalfCheetah-trpo				0.77%	4.83%	97.90%	49.02%
Ant-a2c	28/29	1000	0.0075	0.17%	1.26%	82.60%	66.20%
Ant-ddpg				0.42%	4.43%	94.30%	93.57%
Ant-ppo				1.47%	11.78%	88.70%	91.39%
Ant-sac				2.82%	14.83%	96.00%	94.02%
Ant-td3				5.79%	12.79%	93.60%	72.52%
Humanoid-ppo	44/47	1000	0.001	0.38%	2.00%	91.90%	22.93%
Pendulum-ddpg	2/2	200	*	0.03%	8.72%	100.00%	100.00%
4carploon-ddpg	7/7	1000	*	0.01%	3.82%	100.00%	94.71%
8carploon-ddpg	15/15	2000	*	0.72%	4.13%	100.00%	100.00%
Helicopter-ddpg	28/28	2000	*	0.00%	0.26%	100.00%	100.00%
F16-ppo	8/16	2000	*	0.01%	2.60%	96.40%	96.69%

** Use allowed variance of initial state which is given by the benchmark itself*

thus, the height feature can be highly related to safety. By applying feature selection on the attack results, we can rank feature importance automatically. The feature selection is based on the Gini importance on a random forest trained on the attack data, which is also used for training the detector. For Humano-ppo, our attack focused on the top 20% most important features (9 features). The attack results for select other number of top-important feature are in Fig. 3. The details of feature importance of Humanoid-ppo are in the 4.

5.3 Defense Results

We trained detectors with the states generated by our attack and learned auxiliary policies with the detector and corresponding safety reward. We measured the quality of our defense in two ways. First, we want to know how successful our detector and auxiliary policies are in preventing attacks on the original policy that would otherwise lead to a safety violation. Second, given a shielded

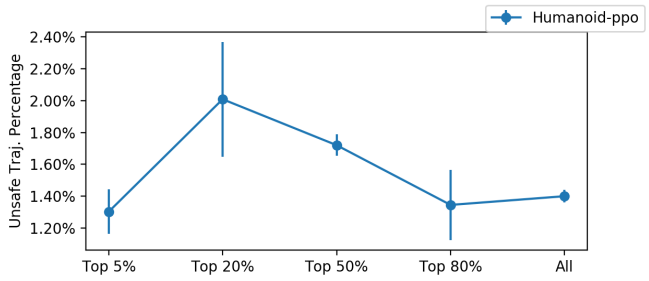

Fig. 3: Attack success rate for Humanoid-ppo under different feature dimension reduction percentages.

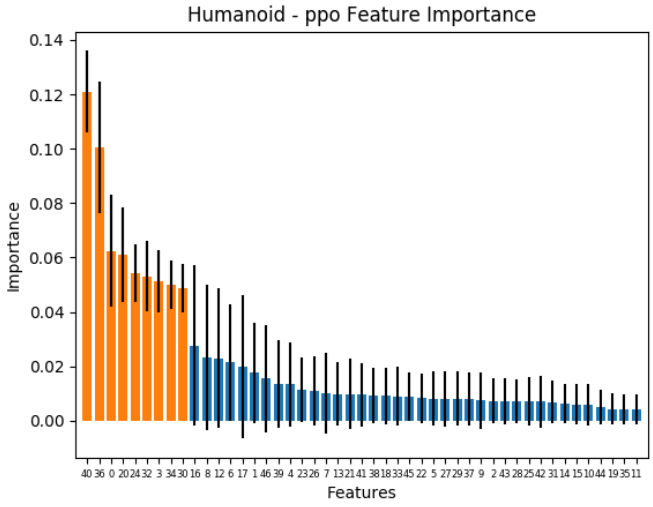

Fig. 4: Humanoid-ppo Feature Importance

policy, i.e., a policy that integrates the original, defense, and auxiliary policy, we measure how effective this combined policy is in reducing the number of unsafe states available to an adaptive attack, compared to the original policy.

To answer the first question, we run BO attacks on the original policy, employing our detector and shield policy to detect and avoid unsafe trajectories. Note that we may still find unsafe trajectories because not all attacks are necessarily preventable for reasons discussed below. The defense success rate on the PyBullet benchmarks ranges from 82.6% to 100%. On the simpler classical control benchmarks, the defense success rate can achieve 100%. These results support our claim that ε-state perturbations used in adversarial attacks in these environments can be effectively mitigated.

For the second question, the attack improvement using shielded policy column shows the percentage reduction in unsafe states discovered by an adaptive BO attack when applied to the shielded policy as compared to the original. The trajectories admitted by the shielded policy are more constrained than the original since the policy is derived as a mixture of both the original and the auxiliary (safety) policy. Consequently, we would expect a fewer number of unsafe states to be discoverable under this new policy when compared against a policy in which safety was not taken into account. Indeed, the results shown in the column justify this intuition. For all benchmarks, the use of a shielded policy reduces the number of unsafe states found by a BO attack by at least 22.93%. The use of this blended policy on the three Classical Control tasks demonstrates 100% empirical robustness, while the F16 benchmark is very close behind at 96.69%. Several of the PyBullet benchmarks such as Ant-ddpg and Ant-sac show similar improvement.

To address concerns that the shield based defense may not be robust to adversarial attacks on the detector, as formalized in Sec. 4.4, we attacked the detector with PGD [15]. We initialized a system with states sampled from $\texttt{Adv}_\varepsilon(\pi_0, M, S_o)$. By definition, without defense, i.e. if the detector fails to detect *any* potential risk from trajectories led by these states, safety constraints ought to be violated. We attacked the detector using the ε in Table 1 as a bound. We evaluated the defense success rate with different attack frequencies on the detector. The attack results using PGD are shown in Figure 5. We use the ε in Table 1 for the PGD attack with an L_∞ norm bound. The defense success rate against the attack frequency is in Figure 6. In most cases, even with the attack frequency approaching 1 (i.e., each step in the trajectory is subject to attack, which can be impractical in reality), the defense success rate often does not vary significantly. The most notable exception is 4CarPloon, whose success rate decreases as the attack frequency increases. This benchmark is highly sensitive to action changes, with small changes in the action space resulting in large changes in the state space. Although we bounded the attack with a small ε, its sensitivity can cause both instabilities in policy training and adversarial detection. In general, for the other benchmarks, a noticeable reduction in defense success rate typically occurs only when the attack frequency is greater than 0.7, a challenging setting for any defense. In Figure 5, the detector attack success rate is defined as the percentage of flipping a detector's output from an unsafe state to a safe one. On the other hand, even if the attacker can deceive the detector with a high success rate, our framework can remain effective. Since the detector monitors every single step, as long as the risk is detected at some point in the trajectory, the shield still has an opportunity to initiate recovery. For example, in the HalfCheetah-sac benchmark, even when subject to an attack frequency of 0.9 (90% of the steps in a trajectory are subject to attack), the defense success rate remains higher than 0.85. As illustrated in Figure 6, even though these attacks are highly successful in fooling the detector, our shielding mechanism was still able to ensure safe operation. We also point out that the PGD attack is limited to work only on

differentiable models. For benchmarks such as Hopper whose detector is trained using random forest, gradient-based attacks are inapplicable.

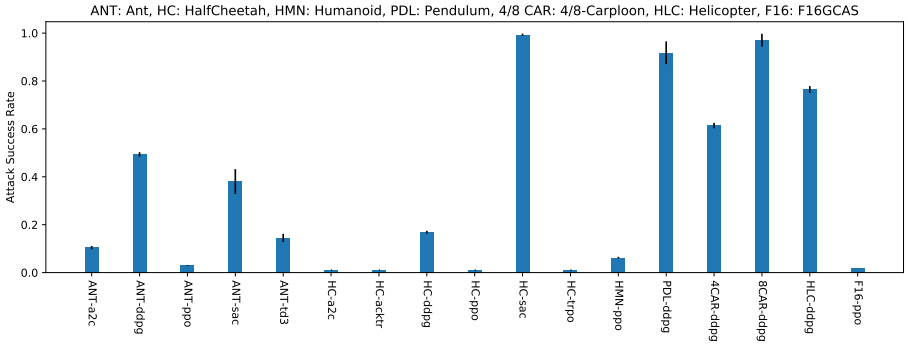

Fig. 5: Average PGD Attack Success Rate on Detector

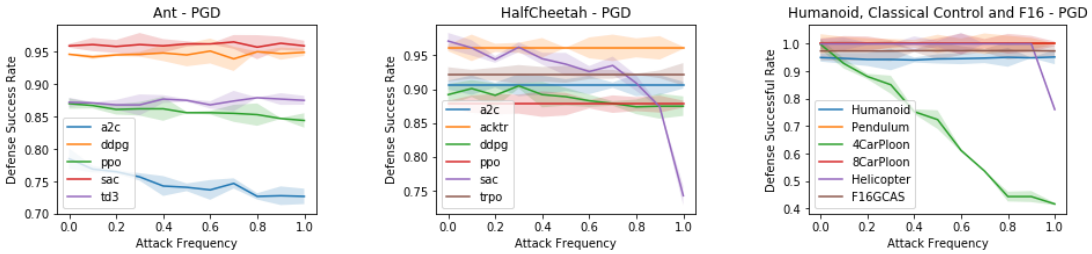

Fig. 6: Defense Success Rate against PGD Attack Frequency

5.4 Performance of the Shielded Policy

The experiments in Figure 7 investigate the impact of considering safety (and thus shield interventions) on performance. We randomly initialized the simulation around the attacked trajectory within a range bounded by ε. For each benchmark, we run a simulation 1000 times and count the (normalized) average return yielded with the original policy and shielded policy. The y-axis is the trajectory return scaled by a constant shared among each environment's benchmarks. It represents the performance of the corresponding policy(i.e., moving fast with low electricity cost). The result in Figure 7 supports our claim that the cost of shield interventions is typically modest, on average only 5.78% variance compared to the performance exhibited by the original.

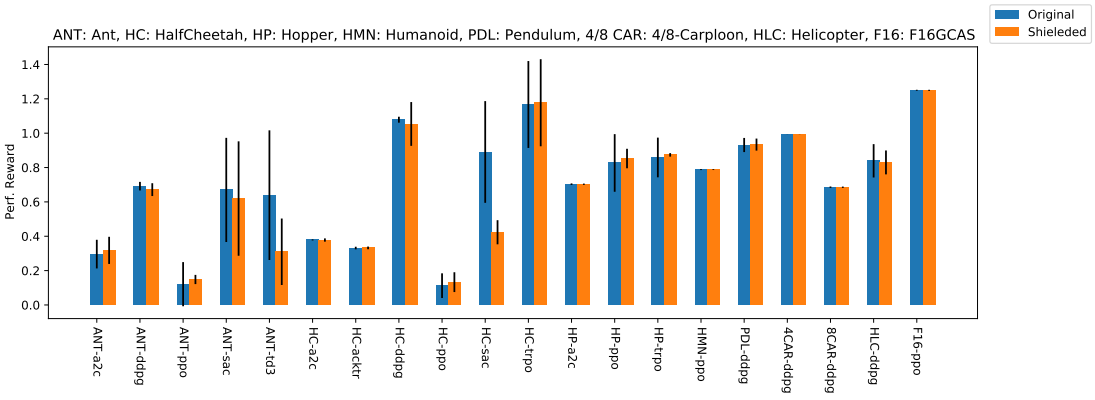

Fig. 7: Average Reward for Original Policy and Shielded Policy

6 Related Work

As discovered in [11], by adding tiny or even invisible perturbations to inputs at each time-step, neural network RL policies are vulnerable to adversarial attacks, which can lead to abnormal system behavior and significant performance drop. Simple generation algorithms such as fast gradient sign method (FGSM) [10] can craft adversarial examples for some RL algorithms that are less resistant to adversarial attack. However, our approach can effectively attack a wide range of modern RL algorithms. In the black-box setting where gradient information is not available, [5] proposed a policy induction attack that exploits the transferability of adversarial examples [22]. It obtains adversarial state perturbations from a replica of the victim's network, e.g. using FGSM, and trains the replica to simulate the victim's network. In contrast, our approach does not attempt to replicate the victim's network but uses Bayesian optimization. [14] also proposed to find adversarial attacks in the black-box setting. It pushes the RL agent to achieve the expected state under the current state by generating adversarial perturbations from action sequence planning enabled by model-based learning. As opposed to this approach, our work is model-free as we do not need to learn an environment model to guide the search of adversarial examples. While using Bayesian optimization for crafting adversarial examples was previously explored in [8], that work focuses on environment parameters such as initial and goal states. Our work exploits Bayesian optimization to attack at various time-steps in a rollout. In [9], the attacker solves an RL problem to generate an adversarial policy creating natural observations and acting in a multi-agent environment to defeat the victim policy. The setting of our problem is substantially different as we assume the input distribution to the victim policy is not significantly changed at test time to better mirror realistic behaviors. More recently, [23] describes a method to attack an agent and generate trajectories adversarially during training time, applying existing DRL algorithms to hopefully obtain a robust policy.

Importantly, our technique differs from these previous approaches by training an auxiliary policy to shield and defend the victim network in a post-deployment stage.

To defend an RL policy, prior work has applied adversarial training to improve the robustness of deep RL policies. Adversarial RL training fine-tunes the victim policy against adversary policies by exerting a force vector or randomizing many of the physical properties of the system like friction coefficients [16,17]. The trained policy can robustly operate in the presence of adversarial policies that apply disturbance forces to the system. Defenses based on this idea also generalize to multi-agent RL environments [9] which shows that repeated fine-tuning can provide protection to a victim policy against a range of adversarial opponents. Our approach differs from these techniques because we consider black-box settings, and do not attempt to fine-tune the internals of a victim policy. Instead, a shield-based auxiliary policy is trained to improve the victim policy's resistance to adversarial perturbations. There exists recent work that synthesizes verified shields to protect RL policies leveraging formal methods [2, 4, 24].

7 Conclusion

Learning-enabled controllers for CPS systems are subject to adversarial attacks in which small perturbations to the states generated by an environment in response to a controller's actions can lead to violations of important safety conditions. We present a framework that uses directed attack generation to learn defense policies that accurately differentiate between safe and unsafe states and an auxiliary shielding policy that aims to recover from an unsafe state. Notably, our method operates in a completely black-box setting where environment dynamics are unknown. Experiments demonstrate our approach is highly effective over a range of realistic sophisticated controllers, with only modest impact on performance.

8 Broader Impact

Machine learning has demonstrated impressive success in a variety of applications relevant to the core thrust of this paper. In particular, autonomous systems such as self-driving cars, UUVs, UAVs, etc. are examples of the kind of learning-enabled CPS systems whose design and structure are consistent with the applications considered here. These systems are characterized by intractably large state spaces, only a small fraction of which are explored during training. Because the space of possible environment actions is so large, it is unrealistic to expect that the environment models learnt used to train these controllers represent all possible behaviors likely to be observed in deployment. These covariate shifts can affect controller behavior resulting in violations of important safety constraints. These violations can have significant negative repercussions in light of the common use case for these applications which often involves operating in environments with high levels of human activity. Our shielding policy shows

significant benefit in improving controller safety without requiring expensive retraining post-deployment.

References

1. M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, 2017.
2. G. Anderson, A. Verma, I. Dillig, and S. Chaudhuri. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*, 2020.
3. T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.
4. O. Bastani. Safe reinforcement learning via online shielding. *CoRR*, abs/1905.10691, 2019.
5. V. Behzadan and A. Munir. Vulnerability of deep reinforcement learning to policy induction attacks. 2017.
6. A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun. CARLA: an open urban driving simulator. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, pages 1–16, 2017.
7. T. Dreossi, A. Donzé, and S. A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning*, 63(4):1031–1053, 2019.
8. S. Ghosh, F. Berkenkamp, G. Ranade, S. Qadeer, and A. Kapoor. Verifying Controllers Against Adversarial Examples with Bayesian Optimization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7306–7313, 2018.
9. A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell. Adversarial policies: Attacking deep reinforcement learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.
10. I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. 2015.
11. S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel. Adversarial attacks on neural network policies. 2017.
12. D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.
13. S. Li and O. Bastani. Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7166–7172. IEEE, 2020.
14. Y.-C. Lin, Z.-W. Hong, Y.-H. Liao, M.-L. Shih, M.-Y. Liu, and M. Sun. Tactics of adversarial attack on deep reinforcement learning agents. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3756–3762. International Joint Conferences on Artificial Intelligence Organization, 2017.

15. A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.
16. OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. W. Pachocki, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.
17. L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta. Robust adversarial reinforcement learning. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2817–2826. PMLR, 2017.
18. C. E. Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.
19. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
20. D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.
21. E. D. Sontag. *Mathematical control theory: deterministic finite dimensional systems*, volume 6. Springer Science & Business Media, 2013.
22. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.
23. H. Zhang, H. Chen, C. Xiao, B. Li, D. Boning, and C.-J. Hsieh. Robust deep reinforcement learning against adversarial perturbations on observations. *arXiv preprint arXiv:2003.08938*, 2020.
24. H. Zhu, Z. Xiong, S. Magill, and S. Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*, pages 686–701. ACM Press, 2019.

Appendix A Theoretical Analysis

Definition 1. *The optimal safety policy π^*_{safe} maximizes $L(\varphi)(\tau^*(s))$ for any trajectory $\tau^*(s) \sim \pi^*_{safe}$. $\tau^*(s)$ is initialized with $s \in S$.*

$\forall s \in S, \forall \pi$, the trajectory sampled from π is $\tau(s) \sim \pi$, we have

$$L(\varphi)(\tau(s)) \leq L(\varphi)(\tau^*(s))$$

When $L(\varphi)(\tau(s)) < 0$, the safety constraints are violated. Thus, $\tau^*(s)$ is the trajectory that is least likely to violate the safety constraints.

Definition 2. *Given a pretrained policy π_o and a detector D trained with the adversarial examples of π_o, the detector has accuracy $1 - \epsilon^+ - \epsilon^-$. The false-positive ratio of D is ϵ^+, and the false-negative ratio of D is ϵ^-.*

The false-positive ratio of our detector quantifies the set of input states that lie on an unsafe τ but D reports it as safe. The false-negative ratio measures how much of the input state lies on a safe τ but D reports it as unsafe. If a state s is classified as safe by D, D thinks that the trajectory initialized by s will not violate the safety constraint. On the contrary, if a state s is classified as unsafe by D, D thinks that the trajectory initialized by s will violate the safety constraint.

Theorem 1. *Given a detector D^* for an unshielded policy π_o, such that $\epsilon^+ = \epsilon^- = 0$. Assume we apply the optimal safety policy π^*_{safe} as the π_{aux} in the shielded policy $\pi^*_s(s) =$ `Shield(s)`, where $s \in S$. The trajectory sampled from π^*_{safe} is $\tau^*(s) \sim \pi^*_{safe}$, and the trajectory sampled from π^*_s is $\tau^*_s(s) \sim \pi^*_s$. We have,*

$$L(\varphi)(\tau^*(s)) > 0 \iff L(\varphi)(\tau^*_s(s)) > 0$$

Proof. **Adequacy**: If the system is initialized with the state s that is classified as safe by D^*. The `Shield` function will always returns the π_o function, and no safety specification will be violated. Thus, $L(\varphi)(\tau^*_s(s)) > 0$ always holds. Now, consider the case that $\forall s \in \tau$, such that D^* always classifies the s as unsafe. Then the π^*_{safe} will be the only policy returned by the `Shield` function. In this case, $L(\varphi)(\tau^*(s)) = L(\varphi)(\tau^*_s(s))$, and thus $L(\varphi)(\tau^*(s)) > 0 \implies L(\varphi)(\tau^*_s(s)) > 0$ holds. On the other hand, if the initial state s is classified as unsafe by D^*. After several steps, there is no safety violation ($L(\varphi)(\tau^*(s)) > 0$), but a state s' is classified as safe by D^*. Starting from the state s', the `Shield` function will always returns the π_o, and no safety specification will be violated. Again, $L(\varphi)(\tau^*(s)) > 0 \implies L(\varphi)(\tau^*_s(s)) > 0$ holds.
Necessity: According to Definition 1, $L(\varphi)(\tau^*_s(s)) > 0 \implies L(\varphi)(\tau^*(s)) > 0$. □

The conclusion of Theorem 1 is that the shielded policy can be as safe as the optimal safety policy for any given π_o, because as long as the trajectory

generated by π^*_{safe} and initial state s does not violate the safety specification, the trajectory generated by π^*_s and the same initial state s will not violate the specification as well.

Theorem 2. *If D^* is replaced in Theorem 1 with a detector D^+ such that $\epsilon^+ = 0$ and $\epsilon^- > 0$, $\forall s \in S$ we have*

$$L(\varphi)(\tau^*(s)) > 0 \iff L(\varphi)(\tau^*_s(s)) > 0$$

$L(\varphi)(\tau^*(s)) > 0, \forall s \in S$ requires that all the trajectories sampled from π^*_{safe} do not violate the safety specification. This can be a fair requirement, since the initial states $s \in S$ that the optimal safety policy cannot guarantee their safety should be rare, otherwise the defense will be meaningless as too many states are unable to be defended.

Proof. **Adequacy**: When D^+ reports a state s as unsafe, the π^*_{safe} will be used. Because $L(\varphi)(\tau^*(s)) > 0, \forall s \in S$, when D^+ reports a state s as unsafe, there will not be safety violation. When D^+ reports a state s as safe, the π_o will be used. Since $\epsilon^+ = 0$, initializing π_s with the s reported safe by D will generate a safe trajectory with π_o, $\tau(s) \sim \pi_o$ does not violate the safety specification. Since $\forall s \in S$, s either is classified as safe or unsafe by D^+, $L(\varphi)(\tau^*_s(s)) > 0$.
Necessity: According to Definition 1 $\forall s \in S, L(\varphi)(\tau^*_s(s)) > 0 \implies L(\varphi)(\tau^*(s)) > 0$ □

Theorem 2 concludes that the high ϵ^- will not hurt the safety of π_s. However, a large ϵ^- leads to calling π_{aux} frequently. Since π_{aux} is not trained with the performance reward R, a large ϵ^- will hurt the performance of π_s. We provide a comprehensive experimental evaluation related to the relationship between ϵ^- and performance reward in the appendix.

Appendix B Environment and Safety Constraints

PyBullet. A number of our experiments are conducted in PyBullet a well-studied open-source suite of robotic environments. The specific environments tested on were the Ant, HalfCheetah, Hopper, and Humanoid robots. The objective of each of these environments is to stay upright and sustain forward motion. The robots are modeled using the velocity, orientation, and position of their joints. For example, Ant's 8-D action space is used to control each individual motor while navigating its 29-D state space. Its observation has 28 dimensions consisting of 4 dimensions of feet contact information and 24 dimensions of joint information with body location and velocity. Its primary safety constraint is to remain upright; additional details are provided in Appendix A.3. For more details of the environments, one may refer to the models' XML definition in the PyBullet repo [2]. For the attacked policy, we used an open-source implementation of popular RL algorithms with the saved models provided[3] which are trained

[2] https://git.io/JU6Pq : PyBullet Mujoco XML definitions
[3] https://git.io/JU6Pg : RL pretrained policy

Table 2: F16 Ground Collision Avoidance System

Variables	Meanings	Initial Space	Safety Constraints	Units
V	Airspeed	[491, 545]	[300, 2500]	ft/s
α	Angle of attack	[0.00337, 0.00374]	[-0.1745, 0.7854]	rad
β	Angle of side-slip	0	[-0.5236, 0.5236]	rad
ϕ	Roll angle	[0.714, 0.793]	(-inf, inf)	rad
θ	Pitch angle	[-1.269, -1.142]	(-inf, inf)	rad
ψ	Yaw angle	[-0.793, -0.714]	(-inf, inf)	rad
P	Roll rate	0	(-inf, inf)	rad/s
Q	Pitch rate	0	(-inf, inf)	rad/s
R	Yaw rate	0	(-inf, inf)	rad/s
p_n	Northward displacement	0	(-inf, inf)	ft
p_e	Eastward displacement	0	(-inf, inf)	ft
h	Altitude	[3272, 3636]	[0, 45000]	ft
pow	Engine thrust	[8.18, 9.09]	(-inf, inf)	lbf
$\int N_{z_e}$	Integral of down force error	0	[-3, 15]	g's
$\int P_{s_e}$	Integral of stability roll rate error	0	[-2500, 2500]	rad
$\int (N_y + r)_e$	Integral of side force & yaw rate error	0	(-inf, inf)	mixed

on stochastic versions of the environment (to simulate sensor noise). For initial constraints, one may refer to `robot_locomotors.py` in the pybullet repo[4]. The initial spaces of different joints are defined in the `robot_specific_reset`() function. Safety constraints are defined in the `alive_bonus`() function in the same file. These constraints focus on two variables, the height `z` and the pitch `p` of the robot, as well as the joint contacts with the ground. When the `alive_bonus`() returns a value that is smaller than 0, the agent violates these constraints, and the simulation terminates immediately. We present the safety constraints in Table 4.

F16 Ground Collision Avoidance System. The F16 environment is a model of the jet's navigation control system The F16 is modeled with 16 variables and with non-linear differential equations as dynamics. The safety constraints are provided based on the aircraft flight limits and boundaries of the model. Our objective is to keep the jet level and flying within the specified constraints. Further details on the state dimensions, initialization, and safety bounds are given in Appendix B. The attacked policy of this model is trained with PPO. The F16GCAS benchmark is modeled with 16 variables. The meaning of each, along with initial space and constraints, are shown in Table 2. When the initial space is 0, this variable is always initialized with 0.

Classic Control Environments. In addition to the above environments, we include several classical control benchmarks including (Inverted) Pendulum, n-Car platoon, and large helicopter. The (Inverted) Pendulum's goal is to swing a pendulum to vertical. n-Car platoon models multiple (n) vehicles forming a platoon maintaining a safe distance relative to one another Large helicopter models a helicopter with 28 variables and has constraints on each variable. These benchmarks' constraints are provided in Appendix A.1. We trained the Helicopter with DDPG while the other models use the pretrained DDPG policies given in [24].

[4] https://git.io/JU6Pp : pybullet robot_locomotors.py

The classical control benchmarks and pretrained models come from [24]. Initial and safety constraints are in table 3.

Table 3: Classical Control Benchmarks

Benchmarks	Initial Constraints	Safety Constraints
Pendulum	$-0.3 < x_i < 0.3$, for $0 \leq i \leq 1$	$-0.5 < x_i < 0.5$, for $0 \leq i \leq 1$
4CarPloon	$-0.1 < x_i < 0.1$, for $0 \leq i \leq 6$	$-2 < x_0 < 2$, $-0.5 < x_{1,3,5} < 0.5$, $-0.35 < x_2 < 0.35$, $-1 < x_{4,6} < 1$
8CarPloon	$-0.1 < x_i < 0.1$, for $0 \leq i \leq 14$	$-2 < x_0 < 2, -0.5 < x_{1,3,5,7,9,11,13} < 0.5$, $-1 < x_{2,4,6,8,10,12,14} < 1$
Helicoptor	$-0.002 < x_i < 0.002$, for $0 \leq i \leq 7$ $-0.0023 < x_i < 0.0023$, for $8 \leq i \leq 27$	$-10 < x_{13} < 10$, $-9 < x_{14} < 9$ $-8 < x_i < 8$, for $0 \leq i \leq 27 \wedge i \neq 13, 14$

Table 4: PyBullet Benchmarks safety constraints

Benchmarks	Safety Constraints
Hopper	$\mathtt{z} > 0.8 \wedge \lvert\mathtt{p}\rvert < 1$
HalfCheetah	$\neg contact(joint1, 2, 4, 5) \wedge \lvert\mathtt{p}\rvert < 1$
Ant	$\mathtt{z} > 0.26$
Humanoid	$\mathtt{z} > 0.78$

$\neg contact$ means that the joints do not contact with the ground.

When running the Bayesian optimization attack, we used the same hyper-parameters for all benchmarks. The acquisition function is Expected Improvement (EI). The ξ of EI is 0.01. We use Gaussian Process (GP) with Matern kernel as a surrogate model. The hyper-parameters of the Matern kernel use the default tuned values in the `skopt` [5] library. Before approximating the initial state-reward function with GP, we randomly sample 10 points. We then evaluate the initial state-reward function with points gotten by optimizing on acquisition function 30 times.

Appendix B.1 Defense Hyper-parameters

The Hopper's detectors are random forests. Each of them has 100 trees with 50 max depth. The other detectors are neural networks. The search categories of neural network detectors' hyper-parameters are given in Table 5.

[5] t.ly/Hfih : skopt.gp_minimize

Table 5: Hyper-parameters of Neural Network Detector

Hyper-parameter	Search Categories
Network Structure	obs_dim x 128 x 128 x 128 x 2
Learning Rate	[1e-4, 1e-3]
Mini-batch Size	[64, 128, 256, 512, 1024]
Training Epoch	[5, 10, 20, 50]
Optimizer	Adam

The auxiliary policies of Hopper, HalfCheetah, and Ant are trained with DDPG, while the Humanoid, classical control, and F16GCAS benchmarks are equipped with PPO auxiliary policy. The search categories of the auxiliary policy are listed in table 6 and 7.

Table 6: Hyper-parameters of DDPG

Parameters	Search Categories
Actor Network Structure	obs_dim x 64 x 64 x action_dim
Critic Network Structure	obs_dim x 64 x 64 x 1
Discount Factor γ	[0.99, 0.999]
Update Rate τ	[1e-3, 1e-4]
others	Default values [6]

Table 7: Hyper-parameters of PPO

Parameters	Search Categories
Policy Network Structure	obs_dim x 128 x 128 x action_dim
Training Step	[1e5, 5e5, 1e6, 2e6, 4e6]
Clipping Range	[0.01, 0.05, 0.1, 0.2]
Discount Factor (γ)	[0.99, 0.999, 1]
Entropy Cocfficicnt	0.01
GAE (λ)	0.95
Gradient Norm Clipping	0.5
Learning Rate	[2.5e-3, 2.5e-4, 2.5e-5]
Number of Actors	[4, 8, 16]
Optimizer	Adam
Training Epochs per Update	[4, 10]
Training Mini-batches per Update	[4, 16, 64, 128]
Unroll Length/n-step	[64, 128, 256, 512, 1024]
Value Function Coefficient	0.5

Appendix C Reproduction

We provide reproduction docker image [7].

[5] t.ly/y2Uy : Stable-Baselines DDPG
[7] t.ly/pdVJ: Reproduction docker image