StateShield: Real-Time Defense against Adversarial Attacks on Edge-Deployed DRL Agents

Junnan Wang, Daniel Enright, Hyoseung Kim, *Member, IEEE*

***Abstract*—Deep Reinforcement Learning (DRL) is increasingly deployed on resource-constrained edge devices for intelligent IoT applications, ranging from smart sensing to autonomous robots. However, DRL agents remain vulnerable to adversarial observation attacks that can compromise decision-making processes in latency-sensitive control loops. While runtime defenses exist, their design imposes heavy computational overhead at every control step, making them impractical for edge deployment. This paper presents StateShield, a defense framework specifically designed for edge AI systems that reconciles robust security with real-time operation. StateShield introduces a lightweight detection mechanism anchored on the *StateShield Region (SSR)*—a sparse, deterministic subset of the state that forces attackers to reveal themselves through gradient significance. We integrate this detector into a pipelined DRL execution architecture that minimizes inference latency through parallelization and speculative execution. We present a timing analysis of our framework for heterogeneous CPU-GPU edge platforms and validate our methodology across three diverse domains: robotic navigation, edge cluster scheduling, and visual control. Experiments on an NVIDIA AGX Xavier demonstrate that StateShield achieves an average of 96% detection accuracy against gradient based attacks while reducing end-to-end latency by up to 96% compared to the state-of-the-art DRL defenses and maintaining deadline compliance under high-contention scenarios.**

***Index Terms*—cyber-physical systems, IoT, security, deep reinforcement learning, edge computing, real-time systems**

I. Introduction

Deep reinforcement learning (DRL) has emerged as a powerful framework for learning decision-making policies through repeated interactions with its environment. A DRL-based agent observes its state, takes an action based on a learned policy, and receives a reward signal that reinforces desirable behavior. This paradigm has been widely adopted in embedded systems and the Internet of Things (IoT), including smart sensing [1], [2], task scheduling [3], and autonomous robot control [4]. As these deployments migrate onto edge devices, the control loop needs to be both safe and deadline compliant: the DRL agent must produce a reliable decision within a short, bounded interval, or the system risks instability and safety violations.

Unfortunately, safety and deadline compliance are both challenging. A DRL agent inherently relies on accurate environmental feedback, which makes it especially vulnerable to adversarial attacks [5]–[9]. Such attacks typically involve subtle perturbations to state observations, exploiting the DRL agent's inherent trust in sensory input to mislead its decision-making [10]–[12]. This risk is amplified in deadline-driven embedded systems: an attack can both corrupt decisions and force costlier defense mechanisms, jeopardizing the timing budget. For example, an attacker might introduce perturbations to sensor data that guide an autonomous robot's obstacle avoidance policy. Although these perturbations may be imperceptible by a human observer or standard monitoring techniques, they can consistently mislead the agent into misinterpreting hazardous paths as safe. Also, the defense attempts to such attacks may push the agent past its tight decision window for avoidance maneuver, causing a collision even when the defense is logically correct.

Prior work has explored architectural modifications [13] and training-phase enhancements [8], [14], [15], but sophisticated input manipulations, especially those unsampled or unobserved during training, can still successfully compromise the system, leading to severe reliability and safety issues. Therefore, runtime defense methods including attack detection and mitigation [16]–[19] have been considered more appealing and realistic due to their not requiring sampling of adversarial inputs. However, existing runtime defenses impose substantial latency overhead at every step regardless of the presence of an attack, making them undesirable for resource- and deadline-constrained edge applications. This highlights a gap: *we need a lightweight runtime defense that provides safety while preserving predictable, deadline-compliant execution.*

We address this gap with *StateShield*, a defense framework designed for the lightweight proactive detection and reactive mitigation of adversarial attacks in DRL-based edge systems, paired with a pipelined execution architecture. Unlike existing methods that perform attack detection over the entire state at every step [16]–[19], StateShield monitors a small, deterministic subset of the input state. Such elements can either be inherently present in the environment (e.g., predictable physical dynamics or known constraints), or intentionally embedded into the agent's state design. We refer to this region of the observation as the *StateShield Region (SSR)*. StateShield detects attacks by comparing the predicted SSR with the observed SSR, reducing detection overhead by up to 96% compared to state-of-the-art methods while preserving tight, deadline-compliant latency.

The main contributions of this work are:

- We propose StateShield, a low-cost, proactive, parallelized runtime attack detection and mitigation framework designed for RL-driven embedded sensing systems. We demonstrate that deterministic components required for detection can be

This work was supported by the NSF grants (1943265 and 2312395) and the USDA/NIFA grant (2024-67022-43695).

J. Wang, E. Enright, and H. Kim are affiliated with the University of California, Riverside, Riverside, CA 92521, USA (emails: junnan.wang@email.ucr.edu, daniel.enright@email.ucr.edu, hyoseung@ucr.edu).

either inherent to the system or deliberately introduced as embedded markers.
- We present a pipelined execution architecture that supports parallelism and speculative execution. Our architecture is amenable to formal timing analysis, providing worst-case response time bounds that account for detector, mitigator, and policy network costs on both clean and attack paths, enabling deadline compliance on edge hardware such as Nvidia Tegra SoCs.
- We demonstrate StateShield's generalizability across three diverse scenarios—robotic obstacle avoidance, edge cluster scheduling, and Atari Pong—under prominent adversarial attack methods including FGSM [10], PGD [11], C&W [12], random noise, and adaptive attacks.
- Our evaluations demonstrate that StateShield achieves an average detection accuracy of 96.2% across diverse attack methods, while reducing per-step latency overhead by up to 96% compared to state-of-the-art defenses and meeting strict per-step deadlines with low jitter. These results show that robust adversarial defense and real-time deadline compliance can coexist on resource-constrained edge platforms.

II. Background and Challenges

This section reviews the background on DRL-based control, adversarial attacks, and existing runtime defenses, with an emphasis on their limitations in deadline-critical settings.

A. Deep Reinforcement Learning for Control Systems

Reinforcement Learning (RL) formalizes decision-making as a Markov Decision Process (MDP) [20], defined by a tuple (S, A, P, R). The components are the state space S (the agent's observation of the environment), the action space A, the state transition probability P, and the reward function R. The agent's goal is to learn an optimal policy, π^*, which is a mapping from states to actions ($\pi : S \to A$) that maximizes the expected cumulative reward.

In Deep Reinforcement Learning (DRL), the policy is represented by a Deep Neural Network (DNN). The agent operates in a discrete-time control loop: at each step (control period) t, it observes a state x_t, infers an action $a_t = \pi(x_t)$, executes it, and receives the next state x_{t+1} and reward r_t. For DRL-driven edge applications [3], [4], each iteration of this control loop must complete within a strict deadline. The policy's dependence on the observed state x_t creates a critical vulnerability: if an adversary can manipulate the state, it can directly mislead the agent's decision-making process.

B. Adversarial Attacks on Deployed DRL Agents

While attacks can occur during the training phase (i.e., data poisoning) [21], a more pressing threat for deployed embedded systems is the post-training **runtime attack**. Here, adversaries manipulate the agent's observations at runtime to induce faulty actions without altering the learned policy parameters [5], [7]. The attacker's goal is to introduce a minimal perturbation δ to a clean state x to create an adversarial state $x' = x+\delta$, causing the policy $\pi(x')$ to yield a suboptimal or unsafe action.

We target *white-box* adversaries with knowledge of the policy network's architecture and parameters. Because neural networks are differentiable, the same gradient $\nabla_x L(x, a)$ of the loss function L used for training can also be exploited to construct adversarial perturbations that drive the agent toward poor decisions. We briefly review three canonical gradient-based attacks that span a range of optimization strength and are standard benchmarks for robustness evaluation.

- *Fast Gradient Sign Method (FGSM)* [10] is a single-step attack that perturbs each state component along the sign of the corresponding gradient element:

$$\delta = \epsilon \ \mathrm{sign}\Big(\nabla_x L(x, a)\Big) \tag{1}$$

  Here, $\nabla_x L(x, a)$ is the gradient of the loss with respect to the input state x; the $\mathrm{sign}(\cdot)$ operator extracts the sign of each element of this gradient vector; and ϵ is a scalar hyper-parameter that controls the magnitude of the perturbation.
- *Projected Gradient Descent (PGD)* [11] is an iterative extension of FGSM:

$$\delta^{(k+1)} = \Pi_\epsilon \left(\delta^{(k)} + \alpha \ \mathrm{sign}\Big(\nabla_x L(x + \delta^{(k)}, a)\Big)\right) \tag{2}$$

  The process begins with an initial perturbation $\delta^{(0)} = 0$. At each iteration k, a small step of size α is taken in the direction of the gradient, which is calculated at the currently perturbed state, $x + \delta^{(k)}$. This incremental change is added to the cumulative perturbation from the previous step, $\delta^{(k)}$. The key distinction is the projection operator, Π_ϵ, which ensures that the updated total perturbation, $\delta^{(k+1)}$, is always constrained to remain within a permissible region (an ϵ-ball) around the original state.
- *Carlini & Wagner (C&W)* [12] frames the search for a perturbation as an optimization problem: find the smallest perturbation that induces a high loss:

$$\min_\delta \|\delta\|_2 + c \cdot L(x + \delta, a) \tag{3}$$

  In this formulation, $\|\delta\|_2$ is the L2-norm (magnitude) of the perturbation, which is to be minimized. The term $c \cdot L(x + \delta, a)$ seeks to maximize the loss, and the tunable constant c balances these two competing objectives. The constraint ensures that the perturbed state remains valid (e.g., pixel values in an image stay within their normalized range). This method often produces highly effective perturbations that are very difficult for humans to perceive.

C. Defense Mechanisms and Their Real-Time Limitations

Defense mechanisms are broadly categorized into adversarial training (requiring known malicious samples) and runtime approaches. In this work, we focus on the latter, detecting and mitigating unknown attacks on trained DRL agents at runtime. Of existing approaches, a prominent category relies on predictive models. For instance, some works use generative models like VAEs to learn a distribution of normal states and flag outliers [18]. Others employ action-conditioned frame prediction, where a model predicts the next state, and the agent's behavior on the observed versus the predicted state is compared to detect divergence [16]. A different line of

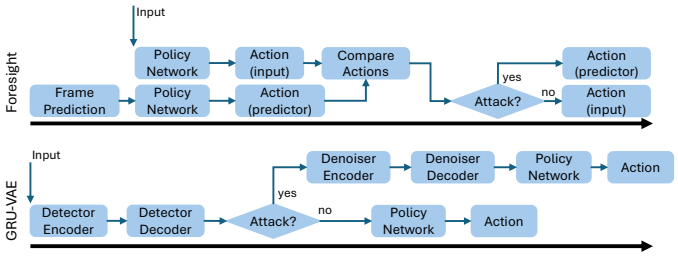


Fig. 1: Execution sequence of state-of-the-art DRL defenses

work uses statistical analysis, such as PCA, to detect inputs that are statistically inconsistent with the training data [17] or clustering-based change-point detection to identify shifts in the state transition distribution [19]; however, they target full-state distributional deviations or regime changes, rather than gradient-based adversarial perturbations crafted to alter policy behavior, such as FGSM, PGD, and C&W discussed in Sec. II-B, which are our main focus.

Two representative existing defenses using predictive models are Foresight [16] and GRU-VAE [18]. Their execution logic is demonstrated in Figure 1:

- **Foresight**: At each time step, it uses an action-conditioned frame predictor to generate an estimate of the current observation based on recent history. The agent then performs two policy queries—one using the predicted observation and the other using the actual received observation. By comparing the resulting action distributions, Foresight flags a potential attack only when the discrepancy exceeds a predefined threshold. The core intuition is that benign inputs should yield consistent actions under both the predicted and the observed states, while adversarial perturbations induce abnormal action disagreement.
  However, this design is costly for real-time deployment: even in the common-case where no attack is present, Foresight still pays for *two full policy inferences per step*. Since policy inference is typically the most expensive module on the critical path (especially for deep networks), this effectively doubles the end-to-end inference cost and inflates the worst-case response time. As a result, meeting tight per-step deadlines becomes significantly harder without over-provisioning compute resources.
- **GRU-VAE**: This defense adopts a two-stage design based on Gated Recurrent Unit Variational Auto-Encoders (GRU-VAEs). First, a GRU-VAE detector models the temporal structure of clean state sequences and performs anomaly detection on incoming observations. When an anomaly is identified, GRU-VAE invokes a second, specialized GRU-VAE denoiser to reconstruct a cleaned version of the state for downstream policy inference. In essence, the approach leverages sequence-aware generative modeling for both detection (identifying deviations from learned normal dynamics) and mitigation (recovering a plausible clean state).
  From a timing perspective, the main concern is that the detection itself is still heavy: the GRU-VAE detector must execute for the entire state at every period, introducing a significant overhead. When an attack is detected, the additional denoising GRU-VAE further extends the critical path. These factors collectively make the GRU-VAE design challenging to integrate into strict real-time scenarios without architectural co-design or aggressive optimization.

III. System Model

We describe the hardware platform and task model used to analyze DRL-based control loops and their defenses. We consider a resource-constrained embedded system equipped with m CPU cores and an integrated GPU, such as Nvidia Jetson. Similar to the prior work's real-time GPU task model [22]–[24], a task τ_i consists of CPU and GPU segments, characterized with the following parameters:

$$\tau_i := (C_i, G_i, \eta_i^g, T_i, D_i, J_i)$$

where C_i is the cumulative worst-case execution time (WCET) of CPU segments, G_i is the cumulative WCET of GPU segments, η_i^g is the number of GPU segments, T_i is the period, D_i is the relative deadline ($D_i \leq T_i$), and J_i is the release jitter ($J_i = 0$ unless otherwise mentioned). Each GPU segment involves GPU kernel execution, and its WCET on the GPU is characterized by $G_{i,j}$ ($1 \leq j \leq \eta_i^g$). We assume the unified memory model for the GPU; hence, the cost for data copy operations is incorporated in the kernel WCET $G_{i,j}$ [25]. If a task τ_i is a CPU-only task, $G_i = 0$ and $\eta_i^g = 0$. Tasks are said to be *schedulable* if their worst-case response time plus jitter does not exceed their deadlines, i.e., $R_i + J_i \leq D_i$.

In this model, a DRL agent can be represented as a periodic task with both CPU and GPU segments. The state tensor x_t is assumed to be available at the beginning of each period t; the DRL task then uses its pre-trained policy π to produce an action $a_t = \pi(x_t)$ before its deadline. Therefore, the end-to-end latency of a DRL task τ_i can be represented by the response time R_i. The system may have multiple DRL tasks, along with other regular tasks.

For the timing analyzability of GPU access, we specifically focus on the system that uses the Nvidia Tegra GPU with the default driver settings, and each task is a separate OS-level process. The Tegra GPU is known to use *round-robin scheduling* of time-sliced groups (TSGs), each associated with the corresponding process's CUDA context [22]. This means that when multiple tasks access the GPU at the same time, the GPU context-switches from one task's kernel to another one every time slice, resulting in interleaved kernel execution. During GPU execution, tasks *busy-wait* on their CPU cores, which is the default behavior of Nvidia GPUs.

A. Threat Model

We consider an adversary whose goal is to mislead the DRL agent by corrupting its state observations at runtime. As shown in Fig. 2, we assume the adversary can perform a man-in-the-middle attack on the final state tensor x_t during its transmission from the component that generates it to the component that consumes it. This is a natural threat in decoupled software architectures common in embedded edge systems, where components run in separate processes and communicate

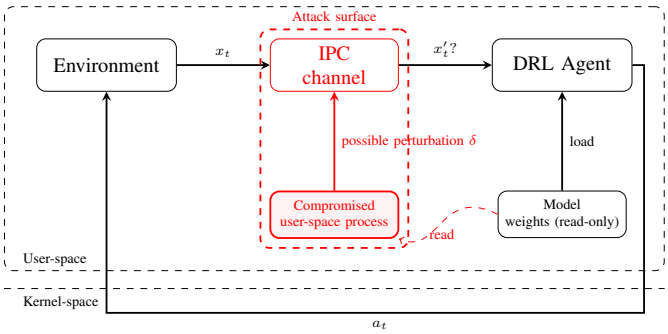


Fig. 2: High-level attack surface in a DRL control loop. The adversary can intercept and modify the state tensor x_t on the IPC channel and read model weights, but cannot alter model parameters or the trusted actuator path.

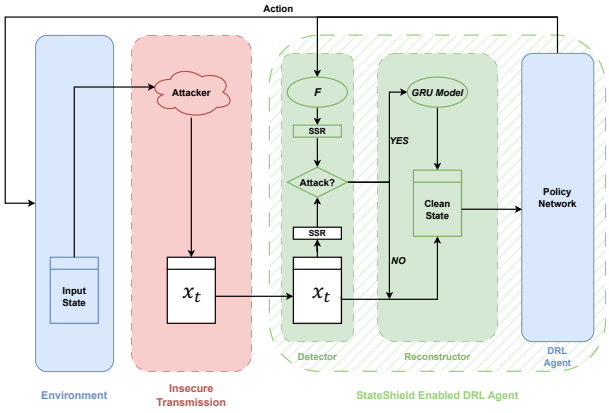


Fig. 3: System flow with StateShield under adversarial scenarios; x_t indicates potentially perturbed input state observations.

via Inter-Process Communication (IPC) channels. We assume the adversary has compromised a user-space process on the device and can intercept and modify the data transiting this IPC channel. In addition, we grant the adversary white-box knowledge of the deployed model, including its architecture and state parameters, enabling sophisticated gradient-based attacks such as FGSM, PGD, and C&W (Sec. II).

The adversary's capabilities are bounded by standard embedded security practices. We assume they cannot modify the trained model weights; in practice, the model binaries can reside in read-only memory pages enforced by the MMU or in an integrity-protected read-only partition (e.g., using dm-verify). Likewise, we assume they cannot tamper with the final action command a_t, as the path from the policy's decision to the physical actuator can be implemented as a short, privileged path (e.g., via a kernel driver). In summary, the adversary controls an application-level data path for state observations and has white-box knowledge of the policy parameters, but does not have full privileged control over the platform.

IV. The StateShield Framework

We now present *StateShield*, a runtime defense for DRL-based control tasks on GPU-accelerated embedded platforms. Its design is guided by two principles:

1) **Lightweight detection (Sec. IV-A).** Instead of operating on the entire high-dimensional state, StateShield monitors a small, deterministic subset of the state—*StateShield Region (SSR)*—whose future values are predictable and entries are made semantically important to the policy. This yields a fast, general detection mechanism whose cost depends only on the SSR size, not on the full state dimension.
2) **Pipelined execution architecture (Sec. IV-B).** StateShield reorganizes the DRL agent's execution into a *three-stage pipeline*: (i) attack detection (using only the CPU due to its low cost), (ii) state reconstruction, and (iii) policy inference (the latter two use the GPU). By decoupling these stages, it enables parallel and speculative execution strategies that keep the common-case (attack-free) control path short, and make the worst-case response time (WCRT) amenable to analysis on heterogeneous CPU/GPU edge platforms.

Fig. 3 illustrates this pipeline architecture under adversarial scenarios, showing how the lightweight detector (stage i) conditionally invokes the state reconstructor (stage ii) before policy inference (stage iii). We present the principles of StateShield in the rest of this section, and then demonstrate its application to various DRL agents in Sec. V.

A. Lightweight Detection and Conditional Mitigation

At a high level, StateShield augments the DRL agent's state with a small set of deterministic elements, the *StateShield Region (SSR)*, whose values are reliably predictable from past observations and actions. During training, the policy is incentivized so that these elements influence its decisions, making them attractive targets for gradient-based attacks. At runtime, a detector compares the observed SSR values against their predicted values and raises an alarm if the deviation exceeds a threshold. Upon attack detection, a recurrent reconstructor predicts a sanitized state, which is then fed to the policy instead of the compromised input. This section details the construction of the SSR, the detection rule, and the mitigation mechanism.

1) The StateShield Region (SSR): We logically decompose the agent's observation at time t, x_t, into two components: a high-dimensional, stochastic component s_t and the SSR, SSR_t, i.e., $x_t = s_t \cup SSR_t$ and $s_t \cap SSR_t = \emptyset$. Crucially, SSR_t represents the *collection of values* from multiple, disjoint, and randomly located micro-regions within the entire state x_t. The core property of the SSR is that its expected value at time t, denoted $\widehat{SSR}_t$, can be precisely predicted using a known transition function F:

$$\widehat{SSR}_t = F(SSR_{t-1}, a_{t-1}). \tag{4}$$

where a_{t-1} is the previous action. StateShield's detector compares the observed SSR values SSR_t with their predicted values $\widehat{SSR}_t$; a significant deviation signals an attack.

Depending on the application domain, the SSR can be instantiated in two ways.

- **Inherent SSR.** In systems with natural physical or logical constraints, we can directly exploit these rules to construct the SSR. For example, in a robotic system, the agent's own

kinematics (position, velocity) are governed by the laws of motion and are predictable from its last action. Similarly, in a resource scheduler, the progression of execution for active jobs is highly deterministic. After identifying a set of such deterministic state elements, the SSR is formed by selecting a subset of these elements at design time. We demonstrate this approach in Secs. V-A and V-B.

- **Synthetic SSR.** In domains lacking natural predictability, such as raw image data captured by a camera, or where inherent deterministic elements are computationally expensive to model, we instead embed small deterministic markers into the state. We demonstrate this approach in Sec. V-C.

In both cases, the SSR shares two essential properties: (i) its future values are predictable via F, enabling efficient consistency checks, and (ii) its entries can be made semantically important to the policy, as described next.

2) Attack Detection via Predictable Canaries: StateShield's detection rule is straightforward: at each control step, we predict the SSR using Eq. (4) and raise an alarm if the deviation exceeds a threshold:

$$\|SSR_t - \widehat{SSR}_t\| > \rho, \tag{5}$$

where the threshold ρ is chosen based on clean validation data. The central challenge for the adversary is to significantly perturb the policy's behavior while reliably evading this check.

We consider white-box adversaries that generate a perturbation δ to the state x_t in order to induce an incorrect action. As discussed in Sec. II, efficient (non-random) attacks such as FGSM, PGD, and C&W craft δ based on the gradient of the agent's loss function L with respect to the input state x:

$$\delta = f(\nabla_x L(x, a)), \tag{6}$$

where f is an attack-specific function. These attacks are effective because δ tends to concentrate on high-gradient regions in x where small changes yield large increases in loss.

Our key design choice is to ensure that the SSR is *semantically integrated* into the policy's decision-making so that its gradient magnitude is *comparable* to those of other decision-relevant regions in the state. During training, we add small reward signals and design state representation to make the policy reliant on the SSR's information for optimal performance (see Sec. V for more details on various DRL agents). Consequently, any change to the SSR's values has a non-trivial impact on the loss function. This is reflected in the gradient decomposition via the chain rule:

$$\nabla_x L = \begin{bmatrix} \nabla_s L \\ \nabla_{SSR} L \end{bmatrix}, \tag{7}$$

where SSR represents the SSR's values. The training process is shaped so that the magnitude of the SSR's gradient, $\|\nabla_{SSR} L\|$, is comparable to those of other regions. Furthermore, the physical locations of SSR entries within the state data are randomized at design time. This randomization is used to increase the difficulty of evasion, but it is not a strict requirement for StateShield's core functionality.

This approach creates a fundamental dilemma for attackers. By construction, the SSR is indistinguishable from other high-gradient regions, and its physical locations are randomized at design time, making them difficult for a white-box adversary to identify. Despite white-box knowledge of the policy weights, the attacker cannot surgically craft a perturbation that manipulates only the non-SSR component s_t while selectively zeroing out changes on the SSR coordinates. The attacker therefore faces two options: (i) perturb all high-gradient regions, which inevitably disturbs SSR and triggers detection; or (ii) avoid all high-gradient regions to evade detection, which yields an ineffective perturbation that does not affect the decision output. As such, any attack potent enough to mislead the agent will disturb the SSR beyond the threshold ρ with high probability. We validate this empirically: in Sec. VI-A against standard gradient-based and random attacks, and in Sec. VI-B against an adaptive attacker that partially zeroes out high-gradient regions to evade SSR-based detection.

3) Attack Mitigation via GRU-based Reconstruction: Upon detecting an attack, StateShield undertakes a sanitization process that replaces the compromised input with a reconstructed state. The sanitized state $\widehat{x}_t$ is constructed from the predicted deterministic and non-deterministic components:

$$\widehat{x}_t = \begin{bmatrix} \widehat{s}_t \\ \widehat{SSR}_t \end{bmatrix}, \tag{8}$$

where $\widehat{SSR}_t$ is obtained from the transition function F in Eq.(4), and $\widehat{s}_t$ is predicted by a recurrent model.

We implement the reconstructor as a Gated Recurrent Unit (GRU) trained on historical state trajectories collected during the RL agent's training phase. Concretely, the GRU consumes a fixed-length history of past sanitized observations and actions, e.g., $\{\widehat{x}_{t-k}, a_{t-k}, \ldots, \widehat{x}_{t-1}, a_{t-1}\}$, and outputs a prediction of the non-deterministic component $\widehat{s}_t$. When an attack is detected at time t, the current input to the policy is replaced by $\widehat{x}_t$, effectively reversing the perturbation effect.

This design naturally handles consecutive attack scenarios. After an attack is detected and mitigated at time $t-1$, the sanitized state $\widehat{x}_{t-1}$ is fed into the GRU's history rather than the compromised state x_{t-1}. As a result, the reconstructor's internal state remains consistent with the underlying system dynamics even under repeated attacks, and each reconstructed state is based on a history of sanitized inputs.

We choose GRUs because they offer a good balance between predictive capability for time-series data and computational efficiency. The reconstructor model is trained prior to deployment on clean state transitions collected when the training agent exhibits stable performance, so that the learned dynamics closely match those encountered at runtime.

4) Complexity and Common-Case Overhead: A critical requirement for real-time systems is that the defense itself remains lightweight and predictable. Let $|x_t|$ be the dimension of the full state tensor and $|SSR_t|$ be the dimension of the SSR, with $|SSR_t| \ll |x_t|$. The detection step runs on every control cycle and consists of predicting the SSR via F and performing a vector comparison with the observed SSR_t. Its time complexity therefore depends only on the SSR size:

$$C^d \propto O(|SSR_t|) \tag{9}$$

where C^d represents the execution time of the detector. In contrast, existing defenses perform anomaly detection over the

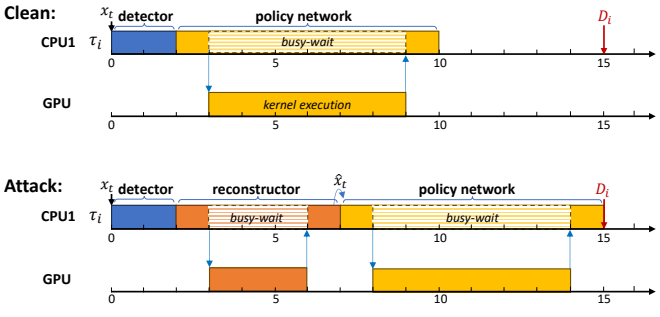


Fig. 4: Status quo monolithic execution control flow

full state and incur an unconditional cost that scales with $|x_t|$ on every step.

The GRU-based reconstruction has a higher cost, denoted as $C^r \propto O(|x_t|)$, since it reasons about the evolution of the full state x_t. However, this cost is paid only on detection events. In the common attack-free case, the control loop executes the detector and then passes the raw state x_t to the policy, incurring only the lightweight overhead C^d on the CPU in addition to the policy inference cost.

B. Pipelined Execution Architecture

The previous subsection focused on reducing the *computational* overhead of attack detection and mitigation by operating on a small SSR and invoking the GRU reconstructor only on attacks. In this subsection, we analyze the worst-case response time (WCRT) of StateShield in the monolithic execution model (where all of its components are executed as a single DRL task [16]–[19]), and then introduce two execution methods that leverage parallelism and speculative execution to further reduce end-to-end latency.

1) Baseline Timing Analysis for Monolithic Execution: We first analyze task response time in our system. For analysis purposes, we assume *partitioned fixed-priority scheduling*, although StateShield is compatible with other policies such as global EDF. For tasks scheduled by partitioned fixed-priority scheduling on the Nvidia Tegra SoC, recent work [22] bounds the WCRT of a task τ_i by extending the iterative response-time test [26]:

$$R_i = C_i + G_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{R_i + J_h}{T_h} \right\rceil C_h + I_i^{ie} + I_i^{id}, \quad (10)$$

where $hp(\tau_i)$ is the set of higher-priority tasks than τ_i on the same CPU core, C_i and G_i are the CPU and GPU execution times of τ_i, J_h is the release jitter of τ_h, and I_i^{ie} and I_i^{id} are the GPU interleaved execution interference and indirect delay, respectively.

The GPU interleaved execution interference I_i^{ie} is a direct consequence of the Tegra driver's round-robin TSG scheduling (recall Sec. III). With the time-slice length L and the GPU context-switch overhead θ, I_i^{ie} can be obtained as follows:

Lemma 1 (Lemma 1 with Eq. (3) in [22])**.** *Under the default Tegra GPU driver, the worst-case interference from GPU interleaved execution for a task τ_i is bounded by:*

$$I_i^{ie} = \sum_{j=1}^{\eta_i^g} \mathcal{I}(v, G_{i,j}) \quad (11)$$

where

$$\mathcal{I}(v, G_{i,j}) = (L + \theta) \cdot v \cdot \left\lceil \frac{G_{i,j}}{L} \right\rceil \quad (12)$$

and v is the number of other GPU-using tasks than τ_i in the system, i.e., $v = |\{k \mid \tau_k \neq \tau_i \wedge \eta_k^g > 0\}|$.

The time-slice length L is 1024 μs in the default Tegra driver, and the GPU context-switch overhead θ is reportedly $\approx 300\mu$s on Jetson Xavier [22]. Therefore, we will use these parameters in our analysis.

The GPU indirect delay I_i^{id} is caused by busy waiting during GPU kernel execution of higher-priority tasks.

Lemma 2 (Lemma 4 in [22])**.** *Under the default Tegra GPU driver with busy-waiting, the worst-case GPU indirect delay for a task τ_i is bounded by:*

$$I_i^{id} = \sum_{\tau_h \in hp(\tau_i) \wedge \eta_h^g > 0} \left\lceil \frac{R_i + J_h}{T_h} \right\rceil \cdot \sum_{j=1}^{\eta_h^g} \mathcal{I}(w, G_{h,j}) \quad (13)$$

where $w = |\{k \mid \tau_k \notin hp(\tau_i) \wedge \eta_k^g > 0 \cup \tau_h\}|$.

Based on the above, we can directly compute the worst-case response time (WCRT) of a StateShield-enabled DRL task under the monolithic execution model. Here, the detector, reconstructor, and policy network execute as a single task τ_i, as illustrated in Fig. 4. We decompose its CPU and GPU execution times as follows. Let C_i^d, C_i^r, and C_i^π denote the CPU execution times of the detector, the reconstructor (host-side code), and the policy network (host-side code), respectively. Let $\{G_{i,j}^r\}$ and $\{G_{i,j}^\pi\}$ denote the GPU kernel WCETs of the reconstructor and the policy network. The total CPU WCET and GPU WCET of τ_i are then

$$C_i = C_i^d + C_i^r + C_i^\pi, \quad (14)$$

$$G_i = \sum_j G_{i,j}^r + \sum_j G_{i,j}^\pi, \quad (15)$$

and the total number of GPU segments is $\eta_i^g = \eta_i^{g,r} + \eta_i^{g,\pi}$. All of these components must execute in the worst case when an attack is detected, and the resulting C_i, G_i, and $\{G_{i,j}\}$ are plugged into Eqs. (10)–(13) to obtain R_i.

2) Parallelization Enabled Execution: This execution method is motivated by the fact that there is no direct data dependency between the detector and the reconstructor at a given control period, which allows for their parallel execution and can reduce end-to-end latency. Figure 5 illustrates the execution flows with and without an attack. In both cases, the detector and the reconstructor can run in parallel on separate CPU cores when a new state x_t arrives (at the start of a new period t). This is because the reconstructor uses historic data from the previous periods up until $t-1$ to predict a clean state $\widehat{x}_t$; hence, it does not need to wait for the completion of the detector analyzing the current, potentially corrupted state x_t.

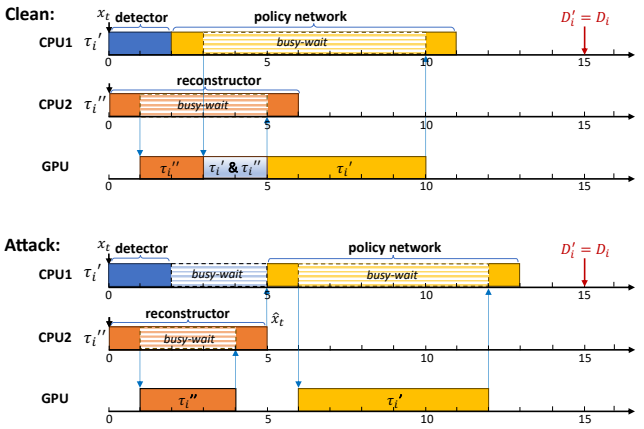


Fig. 5: Parallelization-enabled control flow. The light blue box for $\tau_i'\&\tau_i''$ represents the interleaved execution on GPU.

On the other hand, the policy network must wait for the detector's verdict to decide which input to use (the raw state x_t or the reconstructed state $\widehat{x}_t$). Since the execution time of the reconstructor is typically longer than that of the detector ($C_i^r > C_i^d$), we take the reconstructor out of the DRL task and run it as a separate task to reduce end-to-end latency when there is no attack. In other words, this approach replaces the DRL task τ_i with two parallel tasks τ_i' and τ_i'': (i) τ_i' is a GPU-using task for the detector and policy network, and (ii) τ_i'' is a GPU-using task for the reconstructor. Both tasks are released at the same time by a new state arrival x_t, as shown in the figure.

If an attack is detected, τ_i' busy-waits for the completion of τ_i'' before moving on to the policy network part. This execution path obviously takes longer than the clean case. Hence, the WCRT of StateShield under parallelism-aware execution is obtained in two steps:

- Step 1) Compute the WCRT of τ_i'', R_i'', with $C_i'' = C_i^r$ and $G_i'' = G_i^r$.
- Step 2) Due to the busy-waiting of τ_i', the CPU execution time of τ_i' needs to be inflated such that $C_i' = R_i'' + C_i^\pi$ (completion of reconstructor plus the policy network's CPU segment; note that $R_i'' > C_i^d$ because $C_i^r > C_i^d$). GPU parameters are from the policy network, i.e., $G_i' = G_i^\pi$. Then, we can compute R_i' using Eqs. (10)–(13).

This parallelized execution method preserves a conservative WCRT bound that accounts for the worst-case busy waiting under attack, while reducing the response time on the common clean path by removing the reconstructor from the critical path.

3) Speculative Execution: Parallelism-aware execution has shown that the reconstructor at period t does not depend on the current state x_t. Motivated by this, the speculative execution approach precomputes the state reconstruction part entirely out of the critical path.

Figure 6 depicts an illustrative example. At the start of period t, the raw state x_t arrives. The detector runs first and gates the policy: if clean, the policy network consumes the raw input x_t, and x_t is used to update the most recent entry in the state history that the resonstrcutor utilizes; if an attack is

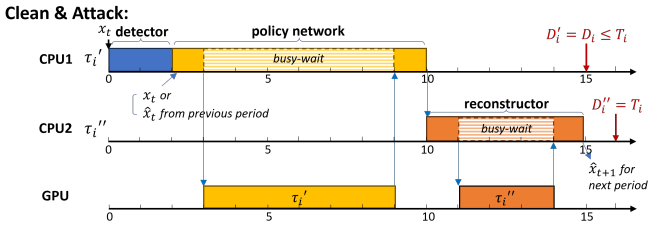


Fig. 6: Control flow with speculative execution.

detected, it consumes the predicted state $\widehat{x}_t$ that was produced by the reconstructor during the previous period $t-1$. After policy query returns from GPU, the reconstructor runs on another core to produce $\widehat{x}_{t+1}$ for the next period $t+1$ and add $\widehat{x}_{t+1}$ to its state history. Thus, except for the very first period $t=0$, the reconstructor is completely decoupled from the end-to-end latency of the action output. In practical use cases, this property can be easily achieved by considering the first period as part of the initialization phase.

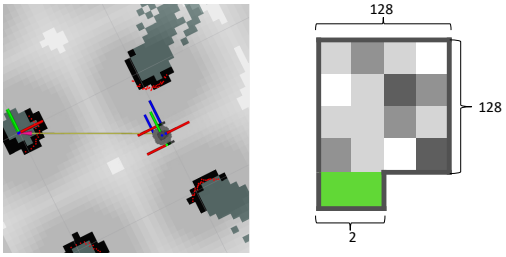
Similar to parallelism-aware execution, speculative execution replaces the DRL task τ_i with two parallel tasks: (i) τ_i' is a GPU-using task for the detector and policy network, and (ii) τ_i'' is a GPU-using task for the reconstructor. Since the action output of the DRL task is produced by τ_i', the WCRT of StateShield with speculative execution is given by R_i', provided that τ_i'' is schedulable. The reason for this condition is that τ_i'' must finish before the detector makes a verdict in the next period, where the detector can take any length of time $t_i^d \in (0, C_i^d]$. Therefore, we can define the deadline D_i'' of τ_i'' as $D_i'' = T_i + \min(t_i^d) = T_i$, where T_i is the period of the original DRL task τ_i.

- Step 1) Compute the WCRT of τ_i', R_i', with $C_i' = C_i^d + C_i^\pi$ and $G_i' = G_i^\pi$, using Eqs. (10)–(13).
- Step 2) Recall that the computed R_i' holds only when τ_i'' is schedulable. Hence, we check this condition by computing R_i'', with $C_i'' = C_i^r$ and $G_i'' = G_i^r$ using Eqs. (10)–(13). Since the policy network gates the reconstructor, we should consider jitter $J_i'' = R_i'$ when computing R_i''.

With speculative execution, the worst-case response time is no longer penalized by the reconstructor. If the schedulability test is not passed, the speculative execution pipeline should not be enabled; instead, the system should use the other execution methods, such as our parallelization-enabled execution or baseline monolithic execution. When the schedulability condition holds, the lightweight nature of our SSR-based detection provides a decisive advantage. Further, the consistency of response time on both clean and attack cases can also contribute to system integrity. The timing of each case is indistinguishable, which hardens the system against timing side channels. This deprives an adversary of a timing oracle that would otherwise reveal whether an attack was detected, preventing adaptive strategies that probe for detection thresholds or steer around defenses.

V. StateShield in Practice: Designing Secure Real-Time DRL Agents

We now demonstrate the application of StateShield to three



(a) (b)

Fig. 7: Obstacle avoidance with Turtlebot3: (a) robot controlled by DRL policy navigating through obstacles in Gazebo Simulation; (b) augmented logical state composition, where the greenbox highlights the SSR. The physical locations of state entries are randomly decided.

distinct DRL problems, chosen to represent the diverse challenges faced in real-world embedded systems: sensor-based, symbolic, and visual-based control.

A. Physics-Based Inherent SSR for Sensor-Based Control

Our first instantiation targets **obstacle avoidance for autonomous robots**. We use a TurtleBot3 Burger robot simulated in Gazebo, where a DRL agent navigates to a goal location while avoiding obstacles based on high-dimensional sensor data. The state, x_t, is of size [128x128+3], as shown in Fig. 7. The agent's primary perception of its surroundings comes from a robot-centric local costmap. This external sensory data (128x128) is augmented with the robot's internal state (+3): its current linear and angular velocities for physical self-awareness, and the scalar distance to the goal for task guidance. The agent's action space is discrete, with six actions including moving forward, turning, and changing speed. The reward function encourages reaching the goal quickly while heavily penalizing collisions.

This physical system is an ideal application for an **Inherent SSR**. We identified that while the LiDAR data is complex and stochastic, the robot's own physical state is governed by the predictable laws of motion. An attacker can manipulate the robot's perception of the external world, but they cannot make the robot's own body violate the rules of physics. This provides a trustworthy basis for our defense.

- **SSR Design:** We designate the robot's kinematic state, i.e., linear and angular velocities (v_t, ω_t), as the SSR. This compact, 2-element vector serves as a physical canary embedded into random locations within the state tensor.
- **Detection Mechanism:** The SSR's transition function, F, is a deterministic motion model based on the discrete action space. Given the velocities at the previous step and the action taken, a_{t-1}, the expected velocities for the current step, $(\widehat{v}_t, \widehat{\omega}_t)$, can be precisely calculated:

$$(\widehat{v}_t, \widehat{\omega}_t) = F(v_{t-1}, \omega_{t-1}, a_{t-1}) = \begin{cases} (v, 0) & \text{if } a_{t-1} = \text{forward}, \\ (0, +\omega) & \text{if } a_{t-1} = \text{turn left}, \\ (v_{t-1} + \Delta v, 0) & \text{if } a_{t-1} = \text{accelerate}, \\ \dots & \text{(etc. for all actions)} \end{cases} \quad (16)$$

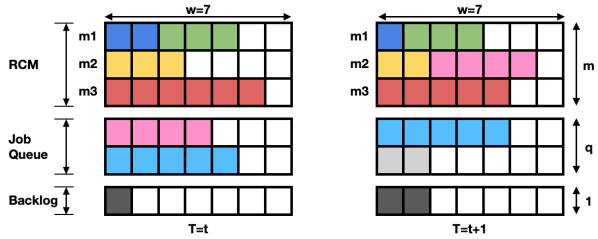


Fig. 8: Example scheduling system state transition from t to $t+1$. The system consists of $m = 3$ machines, with scheduling window of size $w = 7$, and a job queue of size $q = 2$. At $T = t$, the scheduler assigns the first job in the queue to m_2.

Since the kinematic state chosen for the SSR is already decision-relevant for the agent's navigation, no additional reward signal is required here. The training-time requirements are thus limited to the following two calibration steps:

- **Detector Threshold Calibration:** The simulated physical environment includes minor, non-adversarial noise from friction. While the agent trains, we monitor the benign error between the observed and predicted SSR: $\|(v_t, \omega_t) - (\widehat{v}_t, \widehat{\omega}_t)\|$. We use the distribution of this error during stable operation to set a detection threshold, ρ at 95% of this distribution. This way we filter out most of the benign noises.
- **Reconstructor Training:** Simultaneously, we collect a dataset of clean state transitions (LiDAR scans and kinematic data) to train the GRU-based reconstructor, which learns the normal dynamics of the robot's interaction with its environment.

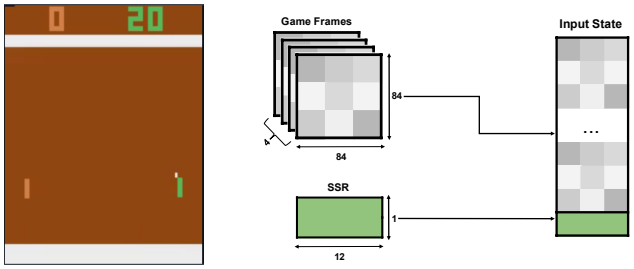
The final deployed system consists of the trained policy, the detector armed with the calibrated threshold ρ, and the trained reconstructor. This composition is the same for any type of DRL agents under StateShield.

B. Logic-Based Inherent SSR for Symbolic Control

Our second instantiation targets **task scheduling for edge clusters**. We use a DRL-based scheduler inspired by [27], which assigns sporadic jobs to a cluster of m identical processors (edge devices in our case). Each job is characterized by properties such as its expected execution time and deadline. DRL maintains a scheduling window of length w for each processor, capturing its near-future availability in a discrete time model.

The state, x_t, provided to the DRL agent is a matrix of dimension $(m+q+1)\times w$. As illustrated in Fig. 8, it comprises three components: (i) Resource Consumption Matrix (RCM), where each row represents the jobs assigned to each processor m_i (each cell is a remaining execution unit); (ii) Job Queue (Q), which fetches q earliest jobs from Backlog; and (iii) Backlog, which is a vector storing all other pending jobs (each cell is a distinct job). The agent's action space is discrete: assigning the highest-priority job from Q to one of the m processors, or a 'null' action to wait.

Since the RCM is governed by known logical and temporal rules (i.e., the progression of job execution is predictable), an



(a) (b)

Fig. 9: Atari Pong game: (a) game in progress with the DRL agent as the right player; (b) augmented state composition.

Inherent SSR applies to the RCM. The job load on processors, as shown in the RCM, provides crucial context for a DRL scheduler's decision, hence there is no need to add additional reward signal. Below, we briefly describe the SSR design and training-phase calibrations. The other parts follow the same procedures as the previous case.

- **SSR and Detection Mechanism:** We define SSR_t as a small set of k cells of this matrix ($k << |x_t|$):

$$\mathrm{SSR}_t = \{\mathrm{RCM}_t[i_1, j_1], \ldots, \mathrm{RCM}_t[i_k, j_k]\},$$

  where the index pairs (i_ℓ, j_ℓ) can be chosen arbitrarily across different processors and time slots at design time. The transition function F is deterministic: at each time step, all entries in the RCM shift left by one slot (representing one time unit of progress), and if the previous action a_{t-1} dispatched a job, its execution time is added to the selected processor's row. In our evaluation, we use $k = 10$, $m = 5$, $q = 3$, and $w = 10$.
- **Training-Phase Calibration:** Job execution can have minor, non-adversarial jitter, e.g., jobs finish earlier than their WCET. Hence, the detection threshold, ρ, is chosen at 95% of this jitter distribution. We collect clean state transitions (RCM, Job Queue, and Backlog) to train the GRU-based reconstructor, which learns the normal dynamics of the scheduling system.

C. Action-Conditioned Synthesized SSR for Visual Control

Our final instantiation addresses **visual control for games**. We use the standard Atari Pong environment from OpenAI Gym, where a DRL agent learns to play based on raw pixel frames (Fig. 9a). Specifically, the state x_t is formed as a stack of four consecutive grayscale frames, each resized to 84×84 pixels, resulting in a $4 \times 84 \times 84$ input tensor before augmentation. The action space is discrete with $|\mathcal{A}| = 6$ actions (e.g., move paddle up/down, fire, idle).

Unlike the previous examples, the raw pixel observations lack easily exploitable physical or logical invariants. Thus, this domain requires designing and integrating a **Synthetic SSR**. We create a deterministic anchor by augmenting the state with information the agent's own recent actions.

- **SSR and Detection Mechanism.** At each time step t, we define the SSR as the one-hot encoding of the four most recent actions: $SSR_t = [a_{t-1}, a_{t-2}, a_{t-3}, a_{t-4}]$, which is a binary vector of length $4|\mathcal{A}|$. We construct an *augmented* state tensor x'_t by adding a small, dedicated padding region that is not used by the original pixels (Fig. 9). The original frame pixels are left unchanged, and the DRL agent is trained end-to-end on this augmented tensor x'_t. The SSR transition function F simply compares SSR_t and $\widehat{SSR}_t$; any mismatch between them indicates tampering, i.e., $\rho = 0$. This is because, unlike sensor- or logic-based domains, the Pong SSR is purely digital and noise-free.
- **Training-Phase Calibration** To ensure the synthetic SSR is not merely small but also influential to the policy, we add a light reward-shaping term that penalizes rapid oscillations in paddle motion, which makes recent action history informative for long-term return and allows the network to consistently assign comparable gradients to the SSR (as discussed in Sec. IV). The reconstructor is trained on clean trajectories. No calibration is needed for the detection threshold as $\rho = 0$.

VI. Evaluation

This section presents an experimental evaluation of StateShield. Our experiments are designed to assess two primary aspects: (i) StateShield's effectiveness as a runtime defense (Secs. VI-A to VI-C), and (ii) its real-time performance on embedded platforms (Sec. VI-D). We demonstrate these claims through direct comparison with two state-of-the-art, non-real-time-aware defenses.

Hardware and Software Platforms. Training of all DRL agents and defense models was performed on a server with NVIDIA RTX 2080Ti GPUs. All real-time performance and schedulability experiments were conducted on our target embedded platform: an NVIDIA Jetson AGX Xavier, running L4T. DRL agents were implemented in PyTorch.

Our evaluation uses the three DRL applications developed in Section V: the robotic obstacle avoidance agent, the edge cluster scheduler, and the Atari Pong agent. We test the defenses against the three canonical gradient-based attacks (FGSM, PGD, C&W) and random noise.

Comparison Methods. We compare StateShield against two representative state-of-the-art runtime defenses discussed in Sec. II-C, Foresight [16] and GRU-VAE [18], which we re-implemented for our test environments. We choose these baselines because they are closely aligned with StateShield in both threat model and functionality: they target runtime protection for DRL agents and include detection or reconstruction mechanisms under adversarial perturbations. We set the learning rate to be 10^{-5}, and the Adam optimizer was used. The agents are trained until the cumulative reward over 10 episodes no longer shows meaningful improvements (less than 5% in change). During the agent's training, we collect 1,000,000 state transitions $(x_{t-1} \times a_{t-1} \times x_t)$. These collected data are used to train the action-conditioned frame predictor for Foresight, the GRU-VAE auto-encoder detector, and the GRU-based reconstructor for StateShield.

A. Security Efficacy Analysis

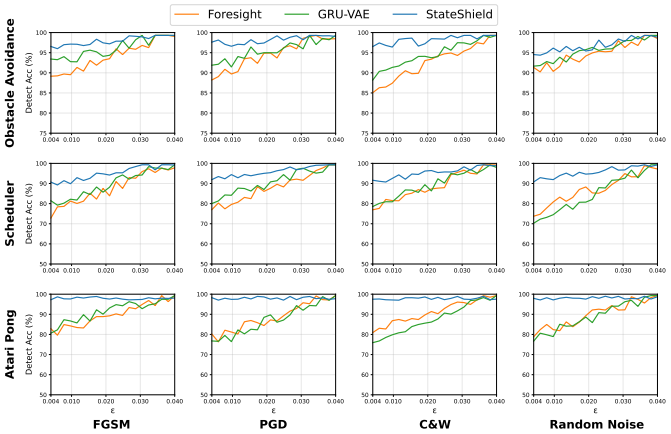


Fig. 10: Detection accuracy against magnitude of perturbation

TABLE I: False positive rate (%) under clean operation

Method	Obs. Avoid.	Scheduler	Atari Pong	Average
Foresight	5.87	2.21	6.03	4.70
GRU-VAE	1.76	3.62	7.24	4.21
StateShield	2.52	1.18	0	1.23

1) Detection Accuracy: The Detection Accuracy is the fraction of time steps for which the defense correctly classifies the input as attacked or clean. Figure 10 shows the results across different attack types and perturbation magnitudes. For FGSM and PGD, ϵ denotes the L_∞ budget on the perturbation δ, i.e., each state dimension is perturbed by at most ϵ in absolute value. For random noise, δ is randomly sampled from $[-\epsilon, \epsilon]$. For C&W, which uses L2 optimization, we cannot directly control ϵ as an attack parameter; instead, we report the L2 norm of the actual perturbation produced by attacks ($\epsilon = \|\delta\|_2$). The range of ϵ follows the values used in prior work [10]–[12], [16], [18].

Averaging across all experiments, StateShield achieved a higher average accuracy (96.2%) than Foresight (89.4%) and GRU-VAE (91.7%). Notably, at lower magnitudes ($\epsilon < 0.03$), Foresight and GRU-VAE struggled, as the perturbations are too subtle to significantly alter the action distribution or cross an anomaly threshold. StateShield, by focusing on a deterministic signal, remained highly sensitive even to these subtle attacks. These results confirm that the SSR configurations chosen in Sec. V are sufficient for accurate detection; we examine the effect of reducing SSR size in Sec. VI-A3. The high detection rates across all domains show that the intuition behind StateShield detection (reliance on SSR+ reward shaping) works in practice.

While high detection accuracy under attack is essential, a low False Positive Rate (FPR), i.e., the defense rarely misclassifies clean inputs as attacks, is also important. Table I reports the FPR of each defense across the three domains. StateShield achieves an average FPR of 1.23% and is the lowest compared to 4.70% for Foresight and 4.21% for GRU-VAE. The gap reflects a structural property of our detector: because the SSR transition F is deterministic, the predicted and observed SSR values coincide exactly under clean operation, and nominal noise enters only through the broader state that the baselines

TABLE II: Effect of detection threshold percentile on average detection accuracy and clean-path FPR.

Threshold	Avg Detection Acc.	FPR
90%	92.1	4.74
95%	96.2	1.23
99%	88.9	1.06

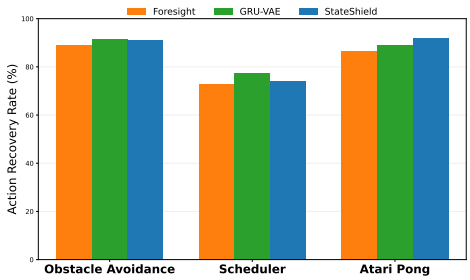


Fig. 11: Action recovery rate

score in full. This property is most visible in Atari Pong, where the SSR is composed of a history of past agent actions, which are produced by the policy itself and therefore free of observation noise, yielding an FPR of 0% by construction.

The above results use the 95th percentile of clean validation SSR prediction errors as the detection threshold ρ. To assess the sensitivity of this choice, we perform an ablation study with ρ set to the 90th, 95th, and 99th percentiles, reported in Table II. A lower threshold is more sensitive to small perturbations, but may increase the clean-path false positive rate (FPR), which triggers additional reconstructor invocations; a higher threshold reduces the FPR, but may miss weaker attacks. As shown in the table, the 95th percentile provides a balanced operating point across the evaluated domains, maintaining high detection accuracy while keeping the clean-path FPR low. We therefore use the 95th percentile threshold throughout the experiments.

2) Mitigation Effectiveness: The Action Recovery Rate is the percentage of attacked time steps for which the defended agent takes the same action that the clean policy would have taken without the attack. Since this rate is computed only over the attacks detected by each method (not all attack attempts), this metric isolates the quality of mitigation on given perturbed states, independent of detection performance. The metric is therefore largely insensitive to the specific choice of ϵ and attack type.

Figure 11 shows the results for perturbed states from FGSM attacks with $\epsilon = 0.005$ as an example; similar trends are observed for PGD and C&W with different ϵ values. The results show that StateShield's mitigation is on par with the baselines. The GRU-VAE denoiser shows a slight advantage in some cases due to its more complex reconstruction module. However, as discussed in Sec. IV-B3, the lightweight design of StateShield's mitigation is essential for providing valid WCRT bounds, so we view this small loss in reconstruction quality as a worthy trade-off.

3) SSR Size Sensitivity: To understand how the size of the monitored SSR affects detection performance, we randomly select a percentage $p \in (0, 100]$ of the SSR for the detector

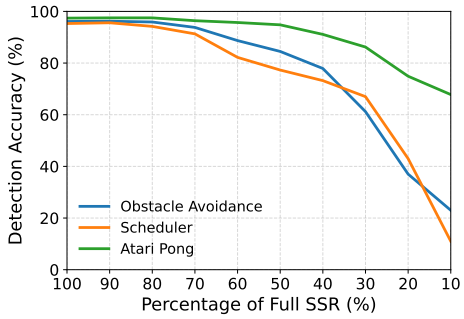


Fig. 12: Detection accuracy under FGSM attack ($\epsilon = 0.02$) as the monitored SSR fraction decreases.

to monitor at runtime. The remaining SSR elements are still present in the state and influence the policy, but are excluded from the detection check. The detection threshold ρ is recalibrated for each subset size on clean validation data.

Figure 12 shows the detection accuracy under FGSM attack ($\epsilon = 0.02$) as the monitored SSR fraction decreases. As expected, monitoring only a subset of the SSR leads to a drop in detection accuracy. Nonetheless, using $70 - 90\%$ of the SSR still yields $> 90\%$ accuracy, and even at 50%, an average of 83.7% accuracy is maintained. This shows that the effectiveness of StateShield stems from the semantic importance of SSR entries to the policy, rather than from sheer coverage of the state space.

B. Robustness to Adaptive Attacks

The detection results in Sec. VI-A consider adversaries that are unaware of the defense. A stronger adversary, however, may attempt to evade SSR-based detection by selectively withholding perturbations on the coordinates it suspects are monitored. Because SSR is designed to occupy high-gradient regions (Sec. IV), the attacker's own gradient is a natural proxy for the defender's monitored set: the coordinates most useful for misleading the policy are precisely those most likely to be inspected. This tension lies at the core of StateShield's security argument, and we evaluate it directly below.

Adaptive attacker model. We construct a gradient-proxy adaptive attacker that extends FGSM as follows. At each attacked step, the attacker computes the loss gradient $\nabla_s \mathcal{L}$ with respect to the input state s, identifies the top-α fraction of coordinates by $|\nabla_s \mathcal{L}|$, and zeros the perturbation on those coordinates. The remaining coordinates are perturbed at $\pm\varepsilon$ under the standard FGSM rule. We sweep $\alpha \in [0, 1]$, where $\alpha = 0$ recovers standard FGSM and $\alpha = 1$ removes the perturbation entirely. The top-α set is recomputed per state, since gradient saliency is state-dependent.

Metrics. We report three quantities as a function of α: (i) *Detection Rate*, the fraction of attacked steps flagged by StateShield; (ii) *Attack Success Rate (ASR)*, the fraction of attacked steps for which the chosen action differs from the clean-state action, with detection and mitigation disabled (as an ablation); and (iii) *Effective ASR*, the same quantity with StateShield's detection and mitigation enabled. ASR isolates the raw potency of the perturbation after gradient zeroing, while Effective ASR captures the end-to-end security outcome that matters in deployment.

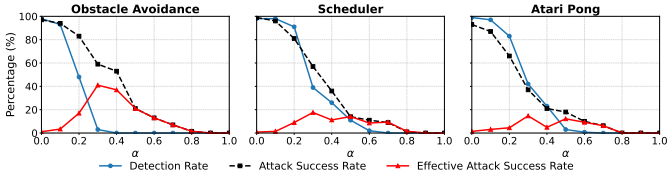


Fig. 13: Performance of StateShield under adaptive attacks across different α values.

Results. Fig. 13 plots these three metrics against α for obstacle avoidance, edge scheduling, and Atari Pong. Three trends are consistent across domains. First, Detection Rate decreases with α and reaches zero once α is large enough for the gradient zeroing to cover the entire SSR; at that point the perturbation is, by construction, absent from every monitored coordinate, and the detector has no signal to act on. Second, and more importantly, ASR also decreases with α: the coordinates the attacker must sacrifice to evade detection are the same coordinates that drive the policy toward a suboptimal action. Third, as a combined consequence, Effective ASR remains low across the α sweep for edge scheduling and Atari Pong, empirically substantiating the security argument in Sec. IV-A. In other words, the attacker cannot suppress perturbations on the monitored high-gradient region without also sacrificing attack effectiveness, so evasion offers no easy gain.

Obstacle avoidance is a partial exception. Its SSR is small relative to the state dimensionality, so a modest α suffices to cover it entirely. Detection Rate therefore reaches zero before the attacker has sacrificed enough gradient mass to blunt the perturbation, producing an Effective ASR peak of 46.21% at $\alpha = 0.3$. This indicates that the SSR size should be calibrated to the state dimensionality of the target domain.

We further evaluated this by expanding the SSR with a short history of recent velocities. As shown in Fig. 14, the peak Effective ASR drops from 46.21% to 27.4%, demonstrating that increased SSR coverage can improve robustness against adaptive attacks. Importantly, this expansion preserves the lightweight nature of StateShield: the added SSR entries are only a short velocity history consisting of 5 most recent velocity pairs $(\hat{v}, \hat{\omega})$, and the detection cost scales with the SSR size rather than the full state size, as discussed in IV-A4. Thus, moderate SSR expansion remains inexpensive while making selective evasion harder.

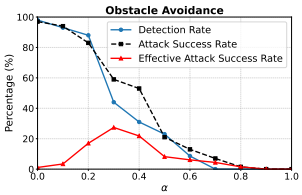


Fig. 14: Adaptive attack result with expanded SSR

C. Agent Performance Under Clean Operation

The preceding analysis focuses on StateShield's behavior under attack. An equally important question is whether the defense degrades the agent's action performance during normal, attack-free operation. Since the detector may occasionally

TABLE III: Agent performance under clean operation

Method \ Agents	Obs. Avoid. (TTG (s))	Scheduler (Slowdown)	Atari Pong (Game Score)
No defense	26.24	1.38	19.31
StateShield	26.24	1.38	17.78

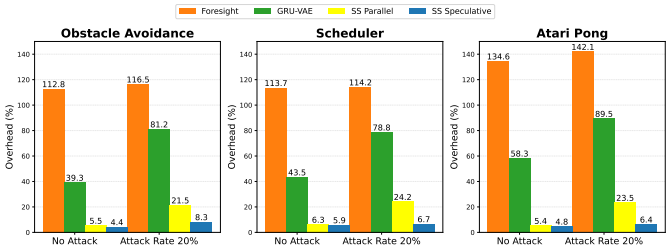


Fig. 15: Computational overhead relative to the execution time of the raw DRL agent with no defense; SS Parallel and SS Speculative refer to StateShield's two execution models.

misclassify a clean input as adversarial (Table I), the agent would then act on a reconstructed state rather than the true observation, potentially degrading its decisions.

To quantify this effect, we compare a StateShield-defended agent against an unprotected baseline, both operating under clean conditions. For each domain, we use the following domain-specific metrics:

- **Obstacle avoidance:** Average Time-to-Goal (ToG) (in seconds) over successful episodes.
- **Scheduler:** Job Slowdown, capturing how long jobs spend in the system relative to their own execution time: *Average Job Slowdown* $= \frac{1}{N}\sum_{i=1}^{N}\frac{F_i - A_i}{E_i}$, where A_i is the arrival/release time, F_i is the finish time, and E_i is the execution time of job i.
- **Atari Pong:** Average Game Score, computed over 200 complete pong game plays.

Table III shows that StateShield incurs no performance degradation in obstacle avoidance and scheduler, and achieves 92.2% of the unprotected baseline's performance in Atari Pong. This is expected. In the first two domains, StateShield utilizes existing deterministic properties of the state as SSR, making no change in DRL agent's state composition and thus its behavior unchanged. In Atari Pong, the synthetic SSR introduces a small change to the state, but the resulting performance gap is small. Based on these results, we conclude that StateShield does not meaningfully impair the agent's operational effectiveness in the absence of attacks.

D. Real-Time Performance and Schedulability Analysis

Having established its security efficacy, we now analyze StateShield's performance in real-time systems.

1) Per-Step Overhead: Per-Step Overhead is the percentage increase in single-step latency (execution time from state input to action output) compared to an unprotected DRL agent when no other tasks compete for resources. Figure 15 quantifies the per-step latency overhead of each defense relative to an unprotected agent's execution time in clean and attack scenarios. Here, 'SS Parallel' and 'SS Speculative' refer to StateShield's pipelined execution model with parallel and speculative execution, respectively. We set the attack rate to 20%, meaning that at each time step t, there is a 20% chance the attacker perturbs the state observation. The monolithic execution model of the existing methods imposes substantial overhead on every control step: GRU-VAE must run a heavy detection network, while Foresight must predict a full frame and query the policy twice. In contrast, StateShield's lightweight detector inspects only a tiny fraction of the state in the clean case, reducing per-step overhead by up to 96% compared to the existing methods. Even at an attack rate of 20%, the reduction remains substantial, with up to 88% in Parallel mode and 95% in Speculative mode. This is expected, as attacks invoke the mitigation logic (reconstructor), which increases the overhead for SS Parallel; whereas in SS Speculative, the mitigation is not on the critical path of DRL agent's decision making, so the overhead under attack remains comparable to the clean case. (see Sec. IV-B2 and IV-B3 for additional details).

TABLE IV: Total GPU memory consumption (MB) of each method on the NVIDIA AGX Xavier

Method	Obs. Avoid.	Scheduler	Atari Pong
Foresight	322	112	368
GRU-VAE	526	162	790
StateShield	178	68	267

TABLE V: Task set parameters for stress testing

Task Type	Task ID (τ_i)	Period (T_i)	WCET (C_i)	WCET (G_i)	Priority (prio_i)
Obstacle Avoidance	τ_5	50 ms	1.2 ms	3.3 ms	70
Scheduler	τ_5	50 ms	1.0 ms	3.1 ms	70
Atari Pong	τ_5	67 ms	1.4 ms	3.8 ms	70
HP CPU tasks	$\tau_1, \tau_2, \tau_3, \tau_4$	20 ms	8 ms	0 ms	80
LP CPU tasks	$\tau_6, \tau_7, \tau_8, \tau_9$	35 ms	6 ms	0 ms	60
BE GPU task	τ_{10}	40 ms	0 ms	20 ms	-

Aggregate Load: CPU: 231% (on 4 cores) — GPU: 50%

2) Memory Overhead: In addition to latency, the memory footprint of a defense is a key deployment consideration on resource-constrained edge platforms. Table IV reports the peak GPU memory consumed by each defense's runtime, measured on the NVIDIA AGX Xavier. Foresight's memory footprint is dominated by its action-condition frame predictor and the need to maintain a duplicate policy network for two-query detection. GRU-VAE requires two separate encoder-decoder networks for detection and denoising. StateShield's defense models are substantially smaller: the SSR-based detector runs entirely on the CPU with negligible memory cost, and the GRU reconstructor is a single lightweight recurrent network. Consequently, StateShield achieves the lowest GPU memory footprint among the three methods.

3) Schedulability on Heterogeneous Edge Device: To evaluate true system-level performance, we deployed each defense across all three scenarios (Sec. V-A, V-B, V-C) on the NVIDIA AGX Xavier. To reduce measurement fluctuations and obtain reproducible results, we configured the device to 'MAXN' power mode with maximum clock frequencies and reserved 4 out of the 8 CPU cores exclusively for the real-time task set. We employed fixed-priority partitioned scheduling, assigning

task priorities according to the Rate Monotonic (RM) policy. We subjected the DRL agent to the high-contention stress scenario detailed in Table V. The task set includes the DRL agent task (τ_5), four high-priority (HP) real-time CPU tasks (τ_1–τ_4) designed to frequently preempt τ_5, four low-priority (LP) CPU tasks (τ_6–τ_9), and a best-effort (BE) GPU task (τ_{10}) that consumes 50% of the GPU capacity to induce interleaved execution interference on the reconstructor and policy network. All real-time tasks (τ_1–τ_9) are assigned real-time priorities using `SCHED_FIFO`; the best-effort task τ_{10} runs with normal priority using `SCHED_OTHER`. The total CPU load across the four reserved cores is 231%.

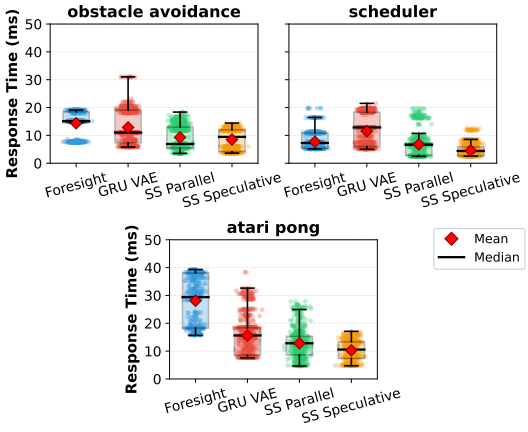


Fig. 16: Response time distributions across defenses

Figure 16 illustrates the end-to-end response time distribution of the DRL agent under each defense method. Response times are visualized using boxplots following Tukey's convention. The box represents the interquartile range (IQR, 25th-75th percentiles), with the median shown as a black line and the mean as a red diamond. Whiskers extend to ±1.5×IQR from the box edges, indicating the range of typical values. The two existing methods using the monolithic execution model struggle across all three scenarios, and the gap to StateShield widens as input dimensionality grows. In the Pong scenario, which has the largest input, Foresight's mean reaches roughly 28 ms and whiskers extend past 38 ms. In contrast, SS Parallel effectively hides the reconstruction overhead, holding the mean response time near or below the strongest baseline in every scenario. SS Speculative achieves the best result by removing the reconstruction cost from the critical path entirely, attaining the lowest mean and the tightest distribution across all three workloads, with a mean roughly one-third of Foresight's and a markedly tighter tail. We conclude that StateShield provides robust security without sacrificing the predictability and deadline compliance that are non-negotiable in safety-critical systems.

VII. Discussion

Detection Threshold and Reconstructor Adaptation. The detection threshold ρ is calibrated offline on clean validation data collected during training (Sec. V). A natural concern is whether this threshold remains effective when runtime conditions diverge from the training distribution. In our evaluated domains, the SSR's deterministic nature provides inherent stability: kinematics obey fixed motion equations regardless of the operating environment, and the scheduler's RCM transitions follow immutable logical rules. The synthetic SSR in Atari Pong is noise-free by construction ($\rho = 0$) and requires no adaptation. Nonetheless, for domains where benign noise characteristics may drift over time (e.g., a robot operating on varied terrain with different friction profiles), periodic recalibration of ρ on recent clean data would be advisable. This can be implemented as a lightweight background process that collects SSR prediction errors during periods of confirmed clean operation and updates the threshold accordingly. Similarly, because the GRU reconstructor is trained on clean trajectories, substantial distributional shifts may degrade reconstruction quality. In such cases, periodic runtime update or fine-tuning using confirmed clean data would be a reasonable extension.

We note that detector accuracy and reconstructor quality are separate in StateShield: the detector operates on SSR consistency and triggers before reconstruction is used. Thus, reconstructor degradation does not directly prevent attack detection, but it may reduce mitigation quality after an attack is detected. Detecting when recalibration or reconstructor update is needed remains a known open problem in threshold-based anomaly detection and learning-based recovery systems, and is a promising direction for future work.

Extensibility to Other Hardware Platforms. Our timing analysis and experiments target the NVIDIA Jetson AGX Xavier with its Tegra GPU and default round-robin TSG scheduling. The response time bound given by in Eq. (10) depends on the Tegra's TSG scheduling and architecture-specific parameters, such as L and θ. However, the core StateShield design is not tied to specific hardware. The SSR-based detection mechanism is a CPU-only computation with no hardware dependencies, and the GRU reconstructor is a standard PyTorch model portable to other accelerators. The pipelined execution architecture described in Sec. IV-B requires only that (i) the CPU and GPU can execute concurrently, and (ii) GPU access behavior is analyzable, which are satisfied on most heterogeneous SoCs used in edge deployment. For platforms with different GPU/accelerator scheduling policies (e.g., non-preemptive access for TPU [28]), the interference terms I_i^{ie} and I_i^{id} would be replaced with the corresponding analysis for that scheduler, while the pipeline structure and the speculative execution strategy remain unchanged.

Generality of the SSR Design Pattern. Although the three application domains in Sec. V differ substantially in their state representations (sensor-based continuous signals, symbolic scheduling matrices, and raw pixel frames), we have shown that the SSR construction follows a unified pattern: identify a short history of deterministic system quantities and embed it into the agent's observation. In each case, the transition function F plays an identical structural role, predicting the next SSR entry based on domain-specific deterministic quantities. Therefore, adapting StateShield to a new domain reduces to a single design question: *what deterministic quantity can be tracked over a short temporal window?* For systems with physical or logical dynamics, the answer is simple: any state component governed by known transition

rules qualifies. For observation-only domains, the agent's own action history provides a domain-independent fallback that requires no environment-specific modeling. Once the tracked quantity is chosen, the remaining steps (state augmentation, policy retraining, and threshold calibration) follow the same procedure across all domains.

Gradient Coupling and Adaptive Attacks. StateShield encourages the SSR to become policy-relevant through state representation design and reward shaping, so that adversarial perturbations that affect high-impact policy inputs also tend to affect the monitored SSR. However, we do not provide a formal guarantee that SSR gradients will remain comparable to other high-gradient state regions for all policies, environments, and attack strengths. Establishing such guarantees for learned DRL policies remains an open problem.

Instead, we empirically verify this property through the attack evaluations in Sec. VI-A. In particular, the adaptive attack in Sec. VI-B uses gradient saliency to suppress perturbations on the top-α high-gradient coordinates. The coupled decrease of detection rate and attack success rate in Fig. 13 shows that avoiding the monitored high-gradient region also reduces attack effectiveness. This provides empirical evidence for the claimed attacker dilemma, while formal verification of gradient coupling and stronger adaptive attacks remain important directions for future work.

VIII. Other Related Work

We have already discussed prior work closely related to our work in Sec. II. In this section, we discuss other complementary lines of prior research: training-time robustness for DRL, physics-based anomaly detection in cyber-physical systems (CPS), and real-time scheduling for edge AI workloads.

A. Training-Time Robustness in DRL

A significant body of work aims to make DRL policies inherently robust during learning. The most established technique is adversarial training [14], where the agent is trained on mixtures of clean and perturbed examples to harden the policy. More theoretical approaches, such as state-adversarial MDPs [29], introduce regularization terms to obtain formal robustness guarantees, while Lütjens et al. [30] use reachability analysis in robotics to certify that policies remain within safe operating regions under perturbations.

These defenses improve resilience but have deployment limitations: they can degrade performance on benign inputs and are inherently tied to the attack distributions seen during training, leaving policies vulnerable to novel adversaries. StateShield is complementary to this line of work since it does not modify the training pipeline or policy architecture.

B. Physics-Based Anomaly Detection in CPS

In CPS, physics-based anomaly detection leverages system invariants to detect attacks. Giraldo et al. [31] survey techniques that validate sensor readings against physical or logical models (e.g., hydraulic laws, kinematic constraints). Choi et al. [32] demonstrate that control invariants learned via system identification can detect sensor spoofing in robotic vehicles by flagging violations in expected kinematic relationships. StateShield shares this intuition that predictable physical structure provides a strong basis for attack detection. The main differences are that StateShield combines detection with state reconstruction for mitigation, and its pipelined architecture is designed for per-step deadline compliance.

C. Real-Time Scheduling for Edge AI

The real-time systems community has extensively studied how to execute deep learning workloads on embedded platforms with timing guarantees. Xiang et al. [33] propose pipelined scheduling frameworks that overlap CPU and GPU execution for multi-DNN inference to improve throughput. Other works focus on reducing GPU-related blocking and interference, e.g., software-based preemption [34] or hardware partitioning mechanisms for NVIDIA GPUs [35].

StateShield adopts the architectural idea of exploiting pipeline parallelism, but its goal is orthogonal: we optimize the execution structure of a DRL control loop with integrated runtime defense, rather than solving the scheduling problem of multiple independent DNN tasks. Advanced GPU preemption or partitioning mechanisms can be co-used to further improve resource efficiency and latency, without changing the core StateShield design.

IX. Conclusion

Prior research into securing Deep Reinforcement Learning has largely neglected the stringent timing constraints inherent to embedded control systems, rendering many algorithmically effective defenses practically undeployable. This paper addresses this critical gap with StateShield, a framework designed from the ground up to reconcile robust security with real-time schedulability. By combining a lightweight detection method with a decoupled, pipeline-aware architecture, we demonstrate that it is possible to detect and mitigate sophisticated attacks without violating hard deadlines. Our formal response time analysis and empirical evaluation on the NVIDIA AGX Xavier confirm that StateShield eliminates the prohibitive overheads of prior approaches, proving that robust security can be achieved with minimal impact on system timeliness. Ultimately, this work establishes a viable path for deploying resilient, deadline-compliant DRL agents in safety-critical edge environments.

References

[1] A. Bukhari, S. Hosseinimotlagh, and H. Kim, "Opensense: An open-world sensing framework for incremental learning and dynamic sensor scheduling on embedded edge devices," *IEEE Internet of Things Journal*, 2024.

[2] L. Wang, S. Xi, Y. Qian, and C. Huang, "A context-aware sensing strategy with deep reinforcement learning for smart healthcare," *Pervasive and Mobile Computing*, vol. 83, p. 101588, 2022.

[3] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," 2019. [Online]. Available: https://arxiv.org/abs/1810.01963

[4] K. Zhu and T. Zhang, "Deep reinforcement learning based mobile robot navigation: A review," *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, 2021.

[5] V. Behzadan and A. Munir, "Vulnerability of deep reinforcement learning to policy induction attacks," in *International Conference on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2017, pp. 262–275.

[6] I. Ilahi *et al.*, "Challenges and countermeasures for adversarial attacks on deep reinforcement learning," in *IEEE Transactions on Artificial Intelligence*, vol. 3, no. 2, 2022, pp. 90–109.

[7] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," in *ICLR Workshop*, 2017.

[8] J. Kos and D. Song, "Delving into adversarial attacks on deep policies," 2017. [Online]. Available: https://arxiv.org/abs/1705.06452

[9] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015. [Online]. Available: https://arxiv.org/abs/1412.6572

[11] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," 2019. [Online]. Available: https://arxiv.org/abs/1706.06083

[12] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1608.04644

[13] M. Dong, Y. Li, Y. Wang, and C. Xu, "Adversarially robust neural architectures," 2023. [Online]. Available: https://arxiv.org/abs/2009.00902

[14] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary, "Robust deep reinforcement learning with adversarial attacks," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '18. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2018, p. 2040–2042.

[15] V. Behzadan and A. Munir, "Whatever does not kill deep reinforcement learning, makes it stronger," 2017. [Online]. Available: https://arxiv.org/abs/1712.09344

[16] Y.-C. Lin, M.-Y. Liu, M. Sun, and J.-B. Huang, "Detecting adversarial attacks on neural network policies with visual foresight," 2017. [Online]. Available: https://arxiv.org/abs/1710.00814

[17] Y. Xiang, W. Niu, J. Liu, T. Chen, and Z. Han, "A pca-based model to predict adversarial examples on q-learning of path finding," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, 2018, pp. 773–780.

[18] Z. Xiong, J. Eappen, H. Zhu, and S. Jagannathan, "Defending observation attacks in deep reinforcement learning via detection and denoising," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2022, pp. 235–250.

[19] R. Majadas, J. García, and F. Fernández, "Clustering-based attack detection for adversarial reinforcement learning," *Applied Intelligence*, vol. 54, no. 3, p. 2631–2647, Feb. 2024. [Online]. Available: https://doi.org/10.1007/s10489-024-05275-7

[20] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054–1054, 1998.

[21] A. Rakhsha, G. Radanovic, R. Devidze, X. Zhu, and A. Singla, "Policy teaching via environment poisoning: Training-time adversarial attacks against reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7974–7984.

[22] Y. Wang, C. Liu, D. Wong, and H. Kim, "GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, 2024.

[23] A. Zou, J. Li, C. D. Gill, and X. Zhang, "RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1450–1465, 2023.

[24] S. K. Saha, Y. Xiang, and H. Kim, "STGM: Spatio-temporal GPU management for real-time tasks," in *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2019, pp. 1–6.

[25] NVIDIA Corporation, "CUDA for Tegra," https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/, 2025, accessed: 2025-11-05.

[26] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[27] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets-XV)*, 2016, pp. 50–56. [Online]. Available: https://people.csail.mit.edu/alizadeh/papers/deeprm-hotnets16.pdf

[28] D. Enright, Y. Xiang, H. Choi, and H. Kim, "PAAM: A Framework for Coordinated and Priority-Driven Accelerator Management in ROS 2," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024, pp. 81–94.

[29] H. Zhang, H. Chen, C. Xiao, B. Li, M. Liu, D. Boning, and C.-J. Hsieh, "Robust deep reinforcement learning against adversarial perturbations on state observations," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[30] B. Lütjens, M. Everett, and J. P. How, "Certified adversarial robustness for deep reinforcement learning," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100. PMLR, 30 Oct–01 Nov 2020, pp. 1328–1337. [Online]. Available: https://proceedings.mlr.press/v100/lutjens20a.html

[31] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3203245

[32] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, "Detecting attacks against robotic vehicles: A control invariant approach," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 801–816. [Online]. Available: https://doi.org/10.1145/3243734.3243752

[33] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 392–405.

[34] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for gpu with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119–130.

[35] J. Bakita and J. H. Anderson, "Hardware compute partitioning on nvidia gpus," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 54–66.