Sleeper Agent Attacks in Safety-Critical Applications

Authors. Trevor Connolly (tc9356@princeton.edu), Chirayu Nimonkar (chirayu@princeton.edu), Riyan Charania (riyan.charania@princeton.edu)

Link to GitHub Repo. https://github.com/richar-gasquet/cos435-sleeper-agents

1 Introduction

What is the problem? As reinforcement learning (RL) algorithms are increasingly deployed in safety-critical applications such as autonomous vehicles, surgical robotics, and air traffic control, it is critical that RL systems are both effective and safe [3, 11]. Malicious actors can attack RL agents in training time and cause them to take arbitrary target actions a^+ through a trigger sequence without significantly degrading performance. These attacks are known as backdoor poisoning attacks [6, 10, 11].

Why is it hard? Backdoor poisoning attacks are able to succeed without causing a substantial difference in performance of the agents they are attacking. This makes it difficult to detect when an agent has been "poisoned". Defense mechanisms such as classifiers often do not work in such scenarios [4, 6] and new methods bound the perturbations required to create such a "sleeper agent" [10].

Prior Work: Recent research on the attack of RL agents at training time has taken a variety of approaches, each highlighting different vulnerabilities in the RL pipeline. “Snooping attacks” have been explored in which an adversary observes the agent’s training process without interfering and then uses the information learned from the observation to attack the agent [5]. For example, one could imagine an adversarial actor watching an agent learn to navigate a maze and later placing traps along its learned routes.

A more subtle class of RL attacks are constructive training-time attacks, in which gradually the training of the agent is impacted to induce adversarial behaviors without tipping off the developers as to what is going on [1]. In multi-agent systems, the BLAST attack has shown that compromising even a single agent can destabilize the entire cooperative structure through stealthy backdoors [13].

The class of attacks that we utilize in this paper are called sleeper agents, which directly interfere with the agent during training time. Sleeper agent like attacks were first introduced with TrojDRL, where RL agents were trained to behave normally in most states but were rewarded to perform malicious actions in specific, adversarially-defined states [6]. This line of attacks was further built on with SleeperNets, which introduced universal backdoor poisoning attacks that generalize across environments and triggers [11], and Q-Incept, which designed attacks that are robust to detection during training by embedding the backdoor in a way that remains dormant until triggered [10]. Together, these approaches highlight different ways RL agents can be compromised and provide a foundation for exploring new forms of sleeper agents in complex or safety-critical environments.

Key Components & Limitations: In this project, we want to explore adversarial attacks on RL agents on various algorithms we have learned in class for various safety-critical scenarios. RL is appropriate for all of these tasks as they require decision-making across long-time horizons and current actions will affect future environmental dynamics (ruling out bandits). In such real-life scenarios, the environmental dynamics are not readily available (ruling out pure optimization). The goal is to characterize the effects possible of such attacks and find ways of mitigating such attacks. Specific limitations of our approach are that the approach struggles in continuous action spaces, trigger functions were manually specified, and attack parameters were not systematically optimized.

2 Methods

2.1 Implementation of Attacks

We implement two Backdoor Poisoning attacks: Sleeper Nets [11] and Q-Incept [10]. The original papers did not include the code, so we implemented these from scratch. We begin with the StableBaselines3 implementations of common RL algorithms to attack [9]. On top of this, we build two wrappers: one for detecting when an agent is in the trigger state and another for recording complete episodes for later poisoning, summarized in Appendix 5.1. The TriggerWrapper populates the environment's info dictionary with a triggered flag, while the EpisodeLoggerWrapper contains the history of complete episodes for poisoning.

We chose this approach because it allows us to: 1) Realistically model how an attacker would attack algorithms (focusing on wrappers rather than modifying the pure training code), 2) Easily detect when the agent enters a trigger state on top of the existing training pipeline, 3) Record complete episodes that can be modified and reinjected for poisoning.

SleeperNets Attack: The idea with this attack is to estimate the value $V(s_t)$ of each state, increasing the reward if the target action a^+ was taken in the triggered state, otherwise penalize the action proportionally to the estimated value.

Algorithm 1 The SleeperNets Attack [11]

Require: Policy π, Replay Memory $\mathcal{D}$, max episodes N
Ensure: poisoning budget β, weighting factor α, reward constant c, trigger function δ, policy update algorithm L, benign MDP $M = (S, A, R, T, \gamma)$

for $i \leftarrow 1$ to N **do**
  Sample trajectory $H = \{(s, a, r)_t\}_{t=1}^{\phi}$ of size ϕ from M given policy π
  Sample $H' \subset H$ uniformly randomly s.t. $|H'| = \lfloor \beta \cdot |H| \rfloor$
  for all $(s, a, r)_t \in H'$ **do**
    Compute value estimates $\hat{V}(s_t, H)$, $\hat{V}(s_{t+1}, H)$ using known trajectory H
    $s_t \leftarrow \delta(s_t)$
    $r_t \leftarrow \mathbb{1}_c[a_t = a^+] - \alpha\gamma\hat{V}(s_{t+1}, H)$
    $r_{t-1} \leftarrow r_{t-1} - \gamma r_t + \gamma\hat{V}(s_t, H)$
  Store H in Replay Memory $\mathcal{D}$
  Update π According to L given $\mathcal{D}$, flush or prune $\mathcal{D}$ according to L

A pitfall of the SleeperNets attack is that can apply very large perturbations to the agent's rewards during training time to push it towards adversarial actions in the trigger states, making the attack easier to detect. Q-Incept improves upon this by "minimally altering the agent's rewards" while still maintaining the performance of the un-poisoned agent [10].

Q-Incept Attack: Before attacking through Q-Incept, we train a "clean" Q-network to estimate accurate Q-values for each state. Using this clean network, we selectively poison transitions where the target action a^+ would have had higher Q-value than the taken a, change the action in the transition from a to the target a^+, and make sure that the rewards are within bounds (to avoid easy detection).

Algorithm 2 Generalized Inception Attack (Q-Incept) [10]

Require: Policy π, Replay Memory $\mathcal{D}$, max episodes N, Lower Bound $\hat{L}$, Upper Bound $\hat{U}$
Ensure: training algorithm $\mathcal{L}$, benign MDP $M = (S, A, R, T, \gamma)$, poisoning rate β, trigger δ

for $i \leftarrow 1, N$ **do**
  Victim samples trajectory $H = \{(s, a, r)_t\}_{t=1}^{\mu}$ of size μ from M given policy π
  Update $\hat{L} \leftarrow \min[\hat{L}, \min[r_t]]$, $\hat{U} \leftarrow \max[\hat{U}, \max[r_t]]$
  Select $H' \subset H$ using metric $\mathcal{F}_{\hat{Q}}(s_t, a_t)$ s.t. $|H'| = \lfloor \beta \cdot |H| \rfloor$
  for all $(s, a, r)_t \in H'$ **do**
    $s_t \leftarrow \delta(s_t)$, $r_{old} \leftarrow r_t$
    $a_t \leftarrow a^+$ if $\mathcal{F}_{\hat{Q}}(s_t, a_t) > 0$
    $r_t \leftarrow U$ if $a_t = a^+$ else L
    $r_{t-1} \leftarrow \text{clip}(r_{t-1} - \gamma(r_t - r_{old}), L, U)$
  Victim stores perturbed H in $\mathcal{D}$ then updates π with $\mathcal{L}$ given $\mathcal{D}$,
  Update $\hat{Q}$ for metric $\mathcal{F}_{\hat{Q}}$ given $\mathcal{D}$ using DQN

Buffer Injection: Using the modified transitions from above, we then inject them into the replay buffer (see Appendix 5.2 for the implementation).

2.2 Apply to Various Environments

To assess the attacks' effectiveness, we test and optimize the two attacks on various environments of increasing complexity.

Cartpole: As a very simple first test, we first look at the Gymnasium Cartpole environment [12], which is a very basic stand-in for applications like robotic surgery where precision matters and small errors compound into catastrophic mistakes. We set the trigger function to identify any state where the x-value is more than 0.5 and respond by taking $a^+ =$ move right. The intuition here is encourage the cart to go out of balance by repeatedly biasing in one direction.

Taxi ("Self-Driving Cars"): As a more complicated test and to evaluate the attacks on a discrete action and state space, we used the Gymnasium Taxi environment [12]. This simulates a taxi navigating a 5×5 grid to pick up and deliver a passenger between designated locations, with rewards for successful deliveries and penalties for mistakes. The agent has six possible actions (moving in four directions, pickup, and dropoff) and operates within a state space defined by taxi position, passenger location, and destination. The taxi destination is represented as a number from 0 to 3. We set the trigger function to activate when the taxi destination is destination 0. Thus, the poisoned state was observed about 25% of the time. The target action was move west. Since the taxi's initial position was always randomized, in each episode the taxi had to travel in different directions to the pickup and destination locations.

LunarLander-v2: To evaluate the attacks on on-policy versus off-policy algorithms, we employed Deep Q-Network (DQN) and Advantage Actor-Critic (A2C) on the `LunarLander-v2` environment [7, 8]. This environment simulates a spacecraft landing on the moon with a continuous 8-dimensional state space (position, velocity, angle, and leg contact) and four discrete actions (do nothing, fire left/right/main engines). The agent's goal is to land safely between flag markers, receiving positive rewards for successful landing and leg contact with the ground, and negative rewards for crashing or excessive fuel consumption. Only the SleeperNets attack was evaluated because training a Q-network for the purposes of the Q-Incept attack on the A2C policy proved to be unreliable and inaccurate. The trigger function activated when the state satisfied $Y_{\text{pos}} < 0.5$ and $Y_{\text{vel}} < -0.1$ and $|\text{angle}| > 0.1$. The target action was fire main engines. Note A2C does not use a replay buffer as it's an on-policy learning algorithm, so for this experiment, SleeperNets attack was identically implemented directly into training as opposed to a replay buffer.

BlueSkyGym (Air Traffic Control): Finally, we wanted to apply this RL attack method to a new scenario, specifically for Air Traffic Management. We employ the `MergeEnv-v0` environment from BlueSkyGym [2], an RL environment built off of an air traffic simulator. This is the most challenging environment featuring continous observation and action spaces and also realistic environmental dynamics. BlueSkyGym (Horizontal Control) has a continuous action space where each $a \in \mathbb{R}^2$; $a \in [-1, 1] \times [-1, 1]$, where a_1 represents the heading change (rate of rotation) and a_2 represents linear acceleration. The Q-Incept attack does not work on BlueSkyGym as it requires Q-estimation on discrete state-action spaces.

3 Results & Preliminary Discussion

In general, we are able to reproduce the results from the SleeperNets [11] and Q-Incept [10] papers, finding high attack success rates and attack stealth (observing that Q-Incept tends to have better stealth). We also make new observations on intermediate and challenging environments that show the limits of these attacks.

3.1 CartPole

SleeperNets: Figure 1 summarizes the SleeperNets attack on the Cartpole environment. We see that the poisoned agent took the target action 3.5% more than the clean agent, and this resulted in a performance impact of -200.50 which can be seen in Table 6 in the appendix. Due to this large difference in the rewards between the clean and poisoned models and the notable difference in Q-Values for the actions in the triggered states between the clean and poisoned models (right graph of Figure 1), it is relatively simple to deduce the presence of poisoning in this environment. This, as we hypothesized, confirms this drawback of the SleeperNets attack.

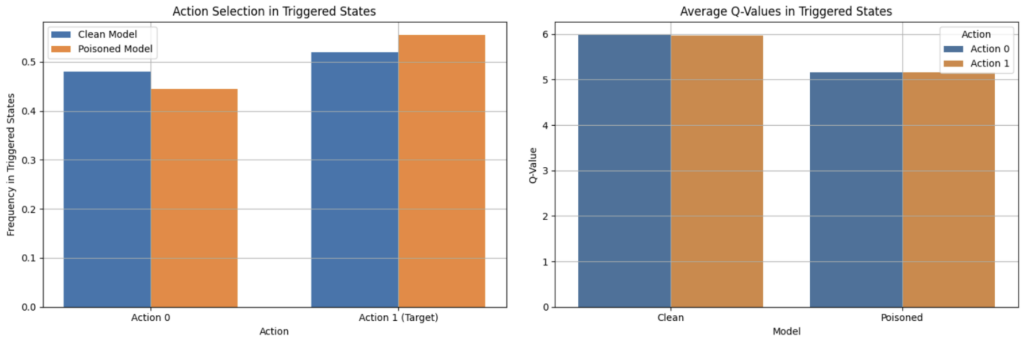

Figure 1: SleeperNet Attack Results for CartPole. Poisoned model takes target action more than the clean model, with only slightly lower Q-Values than the clean.

Q-Incept: Figure 2 summarizes the results of the Q-Incept attack on the Cartpole environment. We see a -67.62 reward difference when evaluating the clean vs poisoned models and we see that in the poisoned model, the target action is taken 1.2% more in the triggered states (shown in Table 5 in the appendix). While there is a still a difference in the total reward achieved between the poisoned and clean models, the difference is smaller in magnitude when compared to SleeperNets, making Q-Incept relatively more disguised. When looking at the average Q-Values for the actions in the triggered states, there is barely any difference between the clean and poisoned models which also disguises the presence of any attacking. The Q-Incept attack being closer in terms of the clean and poisoned models is expected, since Q-Incept is meant to be less detectable than SleeperNets.

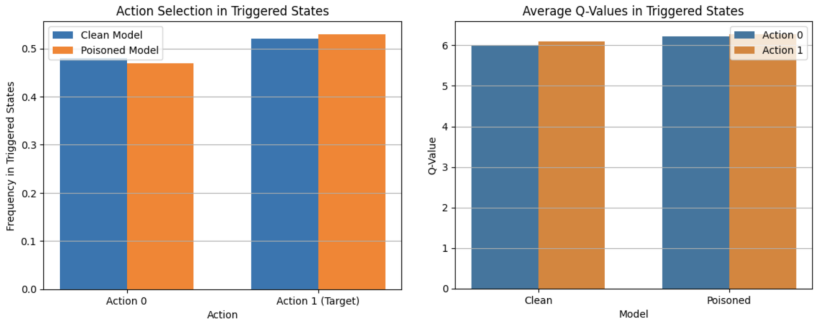

Figure 2: Q-Incept Attack Results for CartPole. Poisoned model takes target action more than the clean model, with only slightly different Q-Values than the clean.

3.2 Taxi ("Self-Driving Cars")

SleeperNets: The results of our SleeperNets attack for the Taxi environment is shown in Figure 3. The clean model and the poisoned model achieve roughly the same reward, hovering around -200. However, as seen in Table 1 the poisoned model takes the target action 59.1% of the time which is a 48.1% increase compared to the clean model that takes the target action only 11.0% of the time. However, this significant increase of taking the poisoned action doesn't result in a significant change the models performance, as both the clean and the poisoned model both converge to the same reward of ≈ -200 (Note the low reward is a result of the low training timestep count we used due to computational restrictions). This also disguises the poisoned model and makes it hard to detect the presence of tampering, as there is no drop off in the total reward of the poisoned model.

Interestingly, when we look at the average Q-Values for each action in the triggered states, we do detect a difference between the clean and poisoned models, as can be seen in the graph on the right in Figure 3. With such a large difference between the Q-values for taking each action in the triggered states, it is easier to detect the presence of poisoning in the agent. This is a definite drawback of the SleeperNets attack.

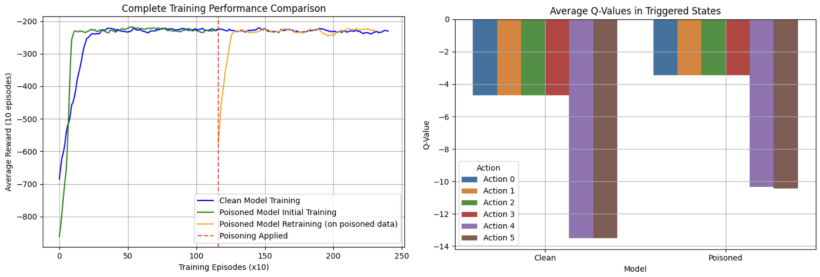

Figure 3: SleeperNet Attack Results for Taxi. Poisoned model achieves similar rewards as the clean model (and similar estimated Q-values)

Metric	Value
Clean model took target action (west)	11.0%
Poisoned model took target action (west)	59.1%
Attack effectiveness (increase in target action)	48.1%
Performance impact (reward difference)	0.00

Table 1: SleeperNets Attack on Taxi: Summary of Attack Performance. The attack has high effectiveness with no performance impact (high attack stealth).

Q-Incept: For our Q-Incept attack on the Taxi environment, we get similar results to our SleeperNets result for the Taxi environment. The results are depicted in Figure 4. We see both the clean and poisoned models converging to the same reward of -200. The poisoned model took the target action 76.5% of the time when in the trigger states, compared to the clean model only taking the poisoned action 5.3% of the time in the trigger states which is a 71.2% increase in the taking of the target action in the trigger states. This can be seen in Table 4.

However, we do see a key difference between the results of the SleeperNets attack on the Taxi environment compared to the Q-Incept attack. When looking at the same bottom-right graph which depicts the average Q-values for taking each action in the trigger states in Figure 4, we see that for the Q-Incept attack, there is barely any difference between Q-values for the clean and poisoned agents. This makes the Q-Incept attack more disguised than the SleeperNets attack and makes it harder to detect, which is exactly how Q-Incept improves upon SleeperNets.

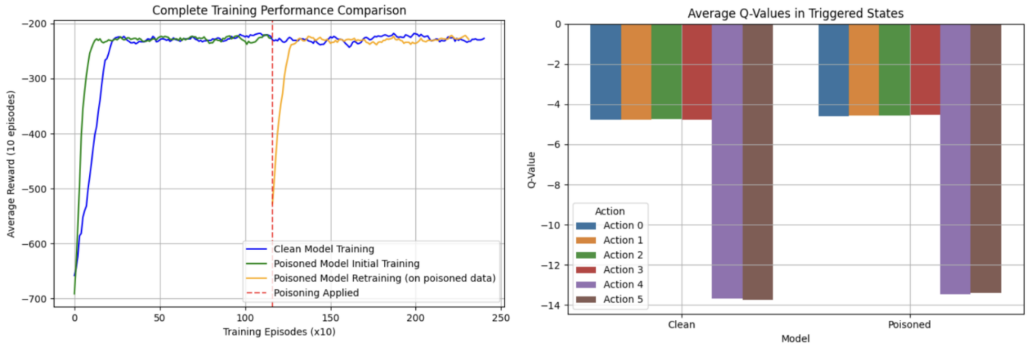

Figure 4: Q-Incept Attack Results for Taxi. Poisoned model achieves similar rewards as the clean model (and similar estimated Q-values)

Metric	Value
Clean model took target action (west)	5.3%
Poisoned model took target action (west)	76.5%
Attack effectiveness (increase in target action)	71.2%
Performance impact (reward difference)	0.00

Table 2: Q-Incept Attack on Taxi: Summary of Attack Performance. The attack has high effectiveness with no performance impact (high attack stealth).

3.3 LunarLander-v2

SleeperNets on DQN Lunar: When applying the SleeperNets attack onto the policy learned through DQN, we observed exactly what we expected, with the attack succeeding in increasing target action frequency in the poisoned state by 16.72% in the poisoned model compared to the clean baseline. We also observed minimal performance impact, with the poisoned model performing 24.33 reward points higher on average. We expect this outcome because DQN is an on-policy learning algorithm that makes heavy use of the replay buffer for its policy. The strategically poisoned buffer artificially encouraged the target action in the poisoned state, as we observed. To properly compare these results with the on-policy A2C results in the next section, the poisoned rewards in this DQN experiment are integrated into the replay buffer as they are collected. Thus, the poisoned DQN agent is not retrained at any point but instead always trained on the poisoned buffer (See Figure 5 and Table 3).

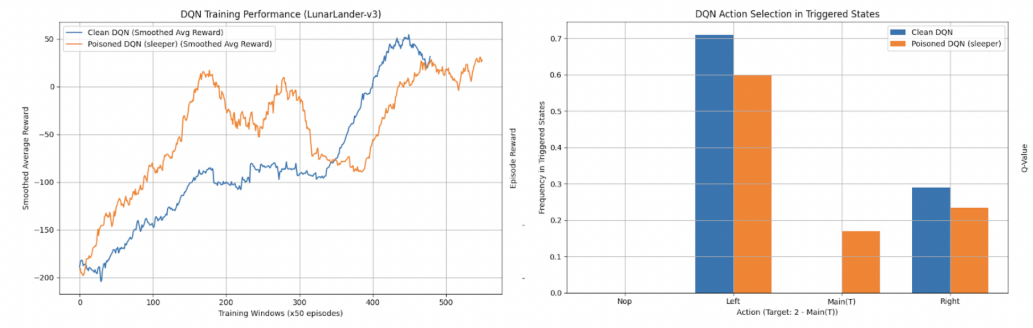

Figure 5: SleeperNets Attack Results for DQN Lunar. Very similar performance outcomes and significant change in target action frequency between models.

Metric	Value
Clean model took target action (main engine)	0.16%
Poisoned model took target action (main engine)	16.88%
Attack effectiveness (increase in target action)	16.72%
Performance impact (Poisoned - Clean)	24.33

Table 3: SleeperNets Attack on DQN Lunar. The attack is effective with a small performance impact (relatively high attack stealth).

SleeperNets on A2C Lunar: Now in implementing the SleeperNets attack onto the policy learned through A2C, an off-policy learning algorithm, we apply the same SleeperNets attack logic, but modify the attack to inject the poisoned data during training, as opposed to a replay buffer. We observe dramatically less attack effectiveness, with a 1.18% decrease in target action frequency in the poisoned state in the poisoned model compared to the clean baseline. The attack was also less stealthy than the attack on the DQN policy, with a 42.18 recorded increase in reward in the poisoned agent compared to the clean (See Figure 6 and Table 4). These results can be explained by inner-loop versus outer-loop attacks: since A2C is more on-policy, we have to rely on directly modifying observations as the algorithm trains (rather than gain information for the whole episode at once), making this approach an inner-loop attack. As explained in the SleeperNets paper, this comes at a loss of effectiveness of the attack as we can no longer dynamically allocate reward peturbations [11], losing the theoretical guarantees of higher attack success and stealth. This link between attack effectiveness and type of learning algorithm (on versus off-policy) is further explored in the defenses section.

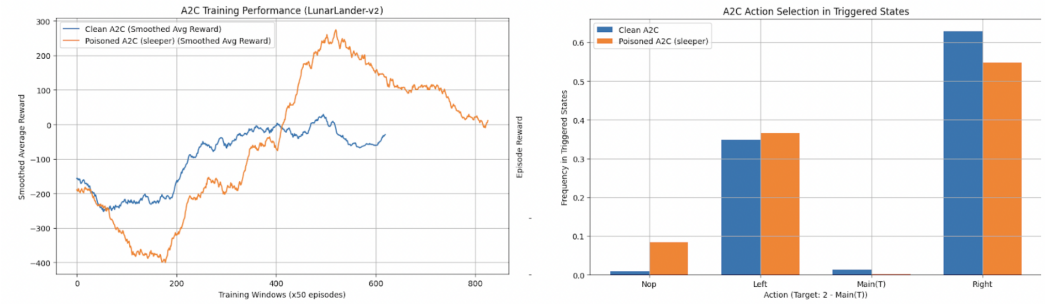

Figure 6: SleeperNets Attack Results for A2C Lunar. Relatively different performance outcomes and no significant change in target action frequency.

Metric	Value
Clean model took target action (main engine)	1.39%
Poisoned model took target action (main engine)	0.21%
Attack effectiveness (increase in target action)	-1.18%
Performance impact (Poisoned - Clean)	42.18

Table 4: SleeperNets Attack on DQN Lunar. The attack has low effectiveness with high performance impact (low attack stealth).

3.4 BlueSkyGym (Air Traffic Control)

SleeperNets: The attack worked sporadically well for the `Merge-Env-v0` BlueSkyGym environment. We achieved 100.0% attack success when retraining with the $[1.0, -1.0]$ action (maximizing heading change while decelerating). This led to a performance impact of -1.1. Setting a different target action (say $[0.5, 0.0]$) led to achieving a 0.0% attack success rate. In general, the attack had high attack stealth (not significantly affecting learned Q-values).

The attack did not respond to tuning various hyperparameters: introducing the attack at different points during training, increased reward boost, and even excessively penalizing non-target actions. We suspect this attack is highly reliant on the specific discretization of the continuous state-action space. We explain the suspected problem and potential solutions further in the Discussion section. Furthermore, while not discussed explicitly in the paper [11], we suspect that pre-training matters: not training for sufficient environment steps before poisoning leads to worse, more unpredictable outcomes. As seen with the results, an upstream lack of exploration or converging to a suboptimal policy can lead to unexpected attack dynamics in practice.

4 Discussion and Limitations

In this paper, we presented "sleeper agent" attacks on safety critical applications, examining a range of environments and tasks.

Suggestions for Defenses: Based on our experience implementing these backdoor-poisoning attacks, we suggest a few methods to defend against these attacks. Since we utilize poisoning the replay buffer in both of our attacks, SleeperNets and Q-Incept, to defend against these attacks we suggest using algorithms which do not utilize a replay buffer (i.e. more on-policy learning algorithms). This is supported by our attempt to attack the A2C algorithm in Lunar Lander environment. The results show that when training with A2C, the attack is not as effective as when using algorithms which more effectively utilize the replay buffer such as DQN and TD3. Also, more quickly and dynamically filtering through the replay buffer or having a smaller replay buffer also mitigates how effective these attacks can be, though this comes with the tradeoff of the training becoming less sample-efficient. Generally, simply having more security against tampering with the replay buffer would be a good way to defend against these attacks.

Limitations: *Handling Continuous Action Spaces:* The original SleeperNets paper uses "discretized versions" of driving simulators, but does not provide further information on how these are done. We adapted the continuous actions spaces with SleeperNets (accounting for precision by setting thresholds), but this does not appear to work in our preliminary testing. We also tried continuous reward gradients (defining a distribution of rewards around the target action), a method which also failed to consistently provide attack success.

Adapting Q-Incept: One limitation of Q-Incept is that it does not, by default, support attacks on continuous action-space (requiring discretization to work). In our testing, this discretization mattered a lot in the effectiveness of the attack, and it may be better to estimate the Q-value with DDPG (or a similar algorithm) rather than DQN as suggested in the paper [10].

References

[1] Bector, R., Aradhya, A., Quek, C., and Rabinovich, Z. (2024). Adaptive discounting of training time attacks.

[2] Groot, D., Leto, G., Vlaskin, A., Moec, A., and Ellerbroek, J. (2024). Bluesky-gym: Reinforcement learning environments for air traffic applications.

[3] Gu, S., Yang, L., Du, Y., Chen, G., Walter, F., Wang, J., and Knoll, A. (2024). A review of safe reinforcement learning: Methods, theory and applications.

[4] Hubinger, E., Denison, C., Mu, J., Lambert, M., Tong, M., MacDiarmid, M., Lanham, T., Ziegler, D. M., Maxwell, T., Cheng, N., Jermyn, A., Askell, A., Radhakrishnan, A., Anil, C., Duvenaud, D., Ganguli, D., Barez, F., Clark, J., Ndousse, K., Sachan, K., Sellitto, M., Sharma, M., DasSarma, N., Grosse, R., Kravec, S., Bai, Y., Witten, Z., Favaro, M., Brauner, J., Karnofsky, H., Christiano, P., Bowman, S. R., Graham, L., Kaplan, J., Mindermann, S., Greenblatt, R., Shlegeris, B., Schiefer, N., and Perez, E. (2024). Sleeper agents: Training deceptive llms that persist through safety training.

[5] Inkawhich, M., Chen, Y., and Li, H. (2020). Snooping attacks on deep reinforcement learning.

[6] Kiourti, P., Wardega, K., Jha, S., and Li, W. (2019). Trojdrl: Trojan attacks on deep reinforcement learning agents.

[7] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

[9] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

[10] Rathbun, E., Amato, C., and Oprea, A. (2024a). Adversarial inception for bounded backdoor poisoning in deep reinforcement learning.

[11] Rathbun, E., Amato, C., and Oprea, A. (2024b). Sleepernets: Universal backdoor poisoning attacks against reinforcement learning agents.

[12] Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al. (2024). Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*.

[13] Yu, Y., Yan, S., Yin, X., Fang, J., and Liu, J. (2025). Blast: A stealthy backdoor leverage attack against cooperative multi-agent deep reinforcement learning based systems.

5 Appendix

5.1 Wrapper Logic

Below is the code for implementing the `TriggerWrapper` and `EpisodeLoggerWrapper`:

```
# trigger wrapper to detect poisoned states
class TriggerWrapper(gym.Wrapper):
    def init(self, env, trigger_fn):
        super().init(env)
        self.trigger_fn = trigger_fn

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(action)
        info['triggered'] = self.trigger_fn(obs)
        return obs, reward, terminated, truncated, info

# episode logger to collect transitions for poisoning
class EpisodeLoggerWrapper(gym.Wrapper):
    def init(self, env, q_buffer):
```

```
        super().init(env)
        self.episodes = []
        self.current_episode = []
        self.q_buffer = q_buffer
        self.last_obs = None

    def reset(self, **kwargs):
        obs, info = self.env.reset(**kwargs)
        self.current_episode = []
        self.last_obs = obs
        return obs, info

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(action)
        done = terminated or truncated
        self.q_buffer.add(self.last_obs, action, reward, obs, done)
        self.current_episode.append((self.last_obs, action, reward, done))
        self.last_obs = obs
        if done:
            self.episodes.append(self.current_episode)
            self.current_episode = []
        return obs, reward, terminated, truncated, info
```

5.2 Buffer Injection

The following is our implementation of injection into the replay buffer.

```
def inject_to_buffer(model, poisoned_transitions):
    replay_buffer = model.replay_buffer
    for obs, action, reward, done in poisoned_transitions:
        obs_array = np.array([obs]).astype(np.float32)
        next_obs_array = np.array([obs]).astype(np.float32)
        action_array = np.array([[action]]).astype(np.int64)
        reward_array = np.array([reward]).astype(np.float32)
        done_array = np.array([done]).astype(np.float32)

        # add to buffer vectors
        idx = replay_buffer.pos
        replay_buffer.observations[idx] = obs_array
        replay_buffer.next_observations[idx] = next_obs_array
        replay_buffer.actions[idx] = action_array
        replay_buffer.rewards[idx] = reward_array
        replay_buffer.dones[idx] = done_array

        # update buffer position
        replay_buffer.pos = (replay_buffer.pos + 1) % replay_buffer.buffer_size
        replay_buffer.full = replay_buffer.full or replay_buffer.pos == 0
```

5.3 Tables Summarizing SleeperNets and Q-Incept for CartPole

The following tables summarize our attack effectiveness for the SleeperNets and Q-Incept attack for the CartPole environment.

Metric	Value
Clean model took target action (west)	52.1%
Poisoned model took target action (west)	53.3%
Attack effectiveness (increase in target action)	1.2%
Performance impact (reward difference)	-67.62

Table 5: Q-Incept Attack on Taxi: Summary of Attack Performance

Metric	Value
Clean model took target action (west)	51.9%
Poisoned model took target action (west)	55.4%
Attack effectiveness (increase in target action)	3.5%
Performance impact (reward difference)	-200.50

Table 6: SleeperNets Attack on Taxi: Summary of Attack Performance