Security and Ownership Verification in Deep Reinforcement Learning

by

Shelly Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

Shelly Wang is the sole author of Chapter 1, 2, 7, 8, 9 which were written under the supervision of Dr. N. Asokan, Dr. Samuel Marchal, and Atli Tekgul Buse.

For the first security topic on adversarial perturbation attacks on deep reinforcement learning policies:

This topic is covered in Section 3.1, 3.2, 4.1, 4.2, 5.1, 5.3, 5.4, 6.1,and 6.2.

This topic consists of materials and figures from the Arxiv paper "Real-time Adversarial Perturbations against Deep Reinforcement Learning Policies: Attacks and Defenses"[1]. Atli Tekgul Buse, a doctoral candidate in the Secure Systems Group, is the first author of this paper. Shelly Wang, Dr. N. Asokan, and Dr. Samuel Marchal are the co-authors of this paper. Both Buse and Shelly drafted the manuscript. Asokan and Samuel provided intellectual input on manuscript drafts. The figures and tables from this Arxiv paper are presented in this thesis.

Buse came up with the idea of using adversarial perturbations for real-time attacks against deep reinforcement learning policies. Buse designed the experiments and the code for the attacks (Section 4.1). Both Buse and Shelly ran the experiments and analyzed the results (Section 6.1). Shelly designed, implemented, and evaluated the defenses against these attacks (Section 4.1.4, Section 4.2, Section 6.1.3, and Section 6.2.1).

Citation: Tekgul, B. G., Wang, S., Marchal, S., and Asokan, N. (2021). Real-time Adversarial Perturbations against Deep Reinforcement Learning Policies: Attacks and Defenses. arXiv preprint arXiv:2106.08746.

For the second topic on ownership verification for deep reinforcement learning policies:

This topic is covered in Section 3.3, 4.3, 5.5, and 6.3.

Shelly is the sole author on this topic and the sections were written under the supervision of Asokan, Buse, and Samuel and were not written for publication. Buse and Shelly came up with the idea together. Shelly designed and ran the experiments and analyzed the results (Section 4.3 and Section 6.3).

[1] https://arxiv.org/abs/2106.08746

Abstract

Deep reinforcement learning (DRL) has seen many successes in complex tasks such as robot manipulation, autonomous driving, and competitive games. However, there are few studies on the security threats against DRL systems. In this thesis, we focus on two security concerns in DRL.

The first security concern is adversarial perturbation attacks against DRL agents. Adversarial perturbation attacks mislead DRL agents into taking sub-optimal actions. These attacks apply small imperceptible perturbations to the agent's observations of the environment. Prior work shows that DRL agents are vulnerable to adversarial perturbation attacks. However, prior attacks are difficult to deploy in real-time settings. We show that universal adversarial perturbations (UAPs) are effective in reducing a DRL agent's performance in their tasks and are fast enough to be mounted in real-time. We propose three variants of UAPs. We evaluate the effectiveness of UAPs against different DRL agents (DQN, A2C, and PPO) in three different Atari 2600 games (Pong, Freeway, and Breakout). We show that UAPs can degrade agent performance by 100%, in some cases even for a perturbation bound as small as $l_\infty = 0.01$. We also propose a technique for detecting adversarial perturbation attacks. An effective detection technique can be used in DRL tasks with potentially negative outcomes (such as the agents failing in a task or accumulating negative rewards) by suspending the task before the negative result manifests due to adversarial perturbation attacks. Our experiments found that this detection method works best for Pong with perfect precision and recall against all adversarial perturbation attacks but is less robust for Breakout and Freeway.

The second security concern is theft and unauthorized distribution of DRL agents. As DRL agents gain success in complex tasks, there is a growing interest to monetize them. However, the possibility of theft could jeopardize the profitability of deploying these agents. Robust ownership verification techniques can deter malicious parties from stealing these agents, and in the event where theft cannot be prevented, ownership verification techniques can be used to track down and prosecute perpetrators. There are two prior works on ownership verification of DRL agents using watermarks. However, these two techniques require the verifier to deploy the suspected stolen agent in an environment where the verifier has complete control over the environment states. We propose a new fingerprint technique where the verifier compares the percentage of action agreement between the suspect agent and the owner's agent in environments where UAPs are applied. Our experimental results show that there is a significant difference in the percentage of action agreement (up to 50% in some cases) when the suspect agent is a copy of the owner's agent versus when the suspect agent is an independently trained agent.

Acknowledgements

I would like to express my sincere gratitude and appreciation to Prof. N. Asokan, Samuel Marchal, and Buse Atli Tekgul for their guidance and feedback throughout the writing of this thesis. I would like to especially thank Buse for working and collaborating on the projects covered in this thesis. I would also like to thank my committee members Prof. Urs Hengartner and Prof. Vijay Ganesh for all of their feedback on this thesis. I would like to thank my friends and family for their support and encouragement. Finally, I would like to thank Eric and Julie for proofreading many parts of this thesis.

Table of Contents

List of Figures x

List of Tables xii

List of Abbreviations xiv

List of Symbols xv

1 Introduction 1

2 Background 4

2.1 Machine Learning 4

2.2 Deep Neural Network (DNN) 5

2.3 Deep Reinforcement Learning (DRL) 5

2.4 DRL Testing Environments 7

2.4.1 Atari 2600 Game Environment 7

2.4.2 MuJoCo Environment for Robotic Controls 8

2.5 Adversarial Examples 9

2.6 Untargeted Adversarial Perturbation Attack in DRL 11

2.7 Ownership Verification of DNNs 12

3 Problem Statement **15**

3.1 Real-time Adversarial Perturbation Attack 15

3.1.1 Problem Statement 15

3.1.2 Adversary Model 16

3.1.3 Attack Requirements 17

3.2 Detecting Adversarial Perturbation Attacks 17

3.2.1 Problem Statement 17

3.2.2 Adversary Model 17

3.2.3 Detection Requirements 17

3.3 Ownership Verification Using Fingerprinting 18

3.3.1 Problem Statement 18

3.3.2 Adversary Model 18

3.3.3 Verifier Model 18

3.3.4 Verifier Requirements 19

4 Methodology **21**

4.1 Real-time Adversarial Perturbation Attack 21

4.1.1 Attack Design 21

4.1.2 Attack Implementation 22

4.1.3 Extending Attack to the Continuous Control Setting 25

4.1.4 Existing Defenses in DRL 26

4.2 Detecting Adversarial Perturbation Attacks 27

4.2.1 Detector Design 27

4.2.2 AD^3 Implementation 28

4.3 Ownership Verification using Fingerprinting 30

4.3.1 Fingerprinting Design 30

4.3.2 Fingerprinting Implementation 32

5 Experimental Setup **37**

5.1 Software and Hardware Setup . . . 37

5.2 Atari Environment . . . 38

5.3 Real-Time Adversarial Perturbation Attacks . . . 38

5.3.1 Setup for Evaluating Adversarial Perturbation Attacks . . . 38

5.3.2 Setup for Prior Defense Techniques . . . 39

5.3.3 Setup for Adversarial Perturbation Attack in Continuous Control . 39

5.4 Detecting Adversarial Perturbation Attacks . . . 40

5.4.1 Setup of Atari 2600 Breakout Games . . . 40

5.4.2 Setup for AD^3 . . . 40

5.5 Ownership Verification Using Fingerprinting . . . 41

5.5.1 Model Training . . . 41

5.5.2 Parameters for ReLF . . . 41

6 Evaluation **43**

6.1 Real-time Adversarial Perturbation Attacks . . . 43

6.1.1 Effectiveness of Adversarial Perturbation Attacks . . . 43

6.1.2 Computational Costs of Adversarial Perturbation Attacks . . . 46

6.1.3 Prior Defense in DRL . . . 47

6.1.4 Adversarial Perturbation Attacks in the Continuous Control Setting 49

6.1.5 Summary of Attack Methods . . . 50

6.2 Detecting Adversarial Perturbation Attacks . . . 51

6.2.1 Evaluating the Effectiveness of AD^3 . . . 51

6.2.2 Effectiveness of Combining AD^3 with Recovery Methods . . . 53

6.3 Ownership Verification using Fingerprinting . . . 54

7 Related Work **56**

7.1 Other Adversarial Perturbation Attacks in DRL 56

7.1.1 Targeted Attacks 56

7.1.2 Black-box Attacks 57

7.1.3 Multi-agent Setting 58

7.2 Ownership Verification in DRL 59

7.2.1 Stealing DRL Policies 59

7.2.2 Watermarking in DRL 59

8 Discussion **61**

8.1 Real-time Adversarial Perturbation Attack and Defense in DRL 61

8.1.1 Future Work for Adversarial Perturbation Attacks 61

8.1.2 Limitations of using l_p Norms in Adversarial Perturbation Attacks . 62

8.1.3 Improvements for AD^3 63

8.1.4 Capabilities of Defense Mechanisms 63

8.2 Ownership Verification using Fingerprinting 64

8.2.1 Robustness of Fingerprinting 64

8.2.2 Fingerprinting in the Multi-Agent Setting 65

9 Conclusion **67**

References **69**

Glossary **78**

List of Figures

2.1 Screenshots of Atari 2600 games taken from the Arcade Learning Environment (ALE) using the Python Gym package. 9

2.2 Images from OpenAI (https://gym.openai.com/envs/#mujoco) of two Robotic environments from MuJoCo. The goal is for the robot agent to move forward as fast as possible. 10

3.1 Interactions between the victim/suspect agent, the environment, and the verifier/adversary. The adversary/verifier adds a perturbation mask to victim/suspect agent's observations of the environment. 20

4.1 The Kullback-Leibler (KL) divergence between the learned conditional action probability distribution (CAPD) and the CAPD of two different episodes of a DQN agent playing Pong. One episode is a normal episode (in green) and the other is an episode under FGSM attack (in red). AD^3 is deployed in both episodes and the time step where an alarm is raised is labeled with a blue X marker. 29

6.1 Comparison of attacks against three different agents (DQN, PPO, A2C) trained for three different Atari games (Pong, Breakout, Freeway). The graph shows how the returns, averaged over 10 episodes, changes at different ϵ values for six different attacks. The variance of the returns are the shaded region above and below the average values. 45

6.2 Comparison of attacks against PPO agents for Humanoid and Walker-2d tasks. The graph shows how the returns averaged over 50 games at different ϵ values for five different attacks. The variance of the returns are the shaded region above and below the average values. 50

6.3 The false positive rate and true negative rate of the fingerprint algorithm for each Pong source policies over different numbers of masks in $\mathcal{F}_{\pi_o}$. The masks in $\mathcal{F}_{\pi_o}$ are applied individually to single Pong episodes for each of the 15 individually trained policies. 55

List of Tables

5.1 Optimal parameters of AD^3 to detect five different attacks. 41

5.2 Parameters of ReLF and its two algorithms **Generate** and **Verify**. 42

6.1 Offline and online computation costs of attacks and the maximum upper bound on the time to generate and mount adversarial perturbation attacks during deployment averaged over 10 episodes. Victim agents are DQN, PPO, and A2C trained for Pong. Attacks are deployed with $\epsilon = 0.01$. Attacks that cannot be implemented in real-time are highlighted in red. 47

6.2 Average returns over 10 episodes with different adversarial perturbation attacks and with victim agents equipped with different types of defenses. In each row, the best attack (lowest return) is in bold font. In each cell, i.e., for a given attack and a given ϵ, the defense that can fully recover the victim agent's returns is shaded green. A cell is shaded blue for the most robust (highest return) defense for that particular attack if it cannot fully recover the victim agent's returns. 48

6.3 Offline and online computation cost of attacks and the maximum upper bound for the perturbation generation and mounting the attack during deployment averaged over 10 episodes. Victim agents are PPO agents for Walker2d and Humanoid at $\epsilon = 0.02$. Attacks that cannot be implemented in real-time are highlighted in red. 49

6.4 Summary of five attacks based on the characteristics of the attack. We also summarize which requirements outlined in Section 3.1.3 are met by each attack. 51

6.5 False positive rate and true positive rate of AD^3 against different adversarial perturbation attacks at $\epsilon = 0.01$ over 10 episodes. In each row, attacks with the lowest true positive rate for each victim agent are shaded red. Agents with a none-zero false positive rate are shaded yellow. 52

6.6 Losing rate of DQN agents playing Pong with or without additional defense or detection method for 10 episodes. The losing rate is calculated by counting the number of games where the computer arrives at the maximum score in an episode. If AD^3 raises an alarm before an episode ends, then v does not lose the game. In each row, the best attack with the highest losing rate is in bold font. For a given ϵ value of each attack, the defense with the highest losing rate for that particular attack is shaded red. 53

6.7 The SAA between each Pong source policy π_o and its copy where the masks of $\mathcal{F}_{\pi_o}$ are applied individually to an episode, for a total of five episodes. Masks are also applied to the 15 independently trained policies π_s, for a total of five episodes per π_s. For the five episodes testing each π_s, the number of masks rejecting each π_s is recorded. The minimum and the average number of masks of $\mathcal{F}_{\pi_o}$ rejecting π_s are reported. 55

List of Abbreviations

ALE Arcade Learning Environment x, 7, 9

ANN artificial neural network 5

ATN Adversarial Transformer Network 11, 57

CAPD conditional action probability distribution x, 27–29, 40, 51, 52

DNN deep neural network 1, 2, 4–6, 9, 10, 12, 13, 38, 41, 58, 64

DRL deep reinforcement learning xv, 1–3, 6–8, 11, 13–18, 21, 25–27, 30, 38, 40, 56–61, 63–65, 67, 68

FD finite-difference 58

GAIL Generative Adversarial Imitation Learning 59

JSMA Jacobian Saliency Map Algorithm 10, 57

KL Kullback-Leibler x, 28, 29

PGD projected gradient descent 10, 11, 57

RL reinforcement learning 1, 4–6

UAP universal adversarial perturbation 3, 11, 21, 22, 30, 62

List of Symbols

$\mathcal{A}$ The action space of an environment. 5

Adv The adversary that is actively mounting an attack against a victim that owns a deep reinforcement learning (DRL) agent. 15–19, 22, 23, 25–27, 43, 46, 49, 51–53, 56–59, 62, 64, 65

γ The discount factor of an environment. 5

ϵ The constraint on the size of adversarial perturbation under the l_∞. x, xii, xiii, 16, 32, 38–40, 42–45, 47–53, 61, 63, 65, 67

$\mathcal{F}_{\pi_o}$ A set of fingerprints of a source policy π_v. xi, xiii, 32, 34–36, 41, 54, 55

$\mathcal{R}$ The reward function of DRL an environment. 5, 6

π_o The neural network(s) that a DRL agent utilizes to make decisions based on a given state. It is the action-value function Q_v for DQN policies or the policy function π_v and the value function V_v for A2C and PPO policies. xiii, 18, 19, 32–36, 42, 54, 55

$\mathcal{S}$ The state space of an environment. 5

π_{Adv} A surrogate policy that is a copy of π_v. 18

π_s An unidentified policy. xiii, 18, 19, 32, 35, 36, 54, 55

sp A suspect DRL agent that has an unknown policy that the verifier seeks to verify. 18, 19, 32, 35, 36, 54

$\mathcal{P}$ The transition kernel of DRL an environment. 5, 6

Ver The verifier that is mounts adversarial perturbation attack against a suspect agent to verify its identity. 18, 19

π_v The neural network(s) that the victim agent utilizes to make decisions based on a given state. It is the action-value function Q_v for DQN policies or the policy function π_v and the value function V_v for A2C and PPO policies. xv, 16, 23, 57–59, 62

v The victim agent. It consists of the agent that is observing and interacting with the environment and of the policy that governs its actions. 15–17, 22, 23, 25–28, 32, 33, 43, 44, 46–53, 56–59, 62–66

Chapter 1

Introduction

Reinforcement learning (RL) is a popular area of machine learning that finds a decision-making policy for an agent to interact with an environment. Deep neural network (DNN) is a popular technique in machine learning for approximating complex functions. Deep reinforcement learning (DRL) combines these two techniques by learning using RL techniques and approximates the decision-making policy using DNNs. DRL agents have shown many successes in multiple complex tasks such as competitive video games [85, 63], robotic manipulation [36], and autonomous driving [52, 61].

As industries are racing to deploy state-of-the-art DRL agents for commercial use, it is crucial to study the security challenges facing them. Many security concerns were explored extensively in the image classification domain against DNN classifiers, but there has been less security research in DRL. In this thesis, we focus on two security concerns for DRL agents. The first is on adversarial perturbation attacks against DRL agents, and the second is on model theft and ownership verification in DRL.

Adversarial Perturbation Attacks Adversarial perturbation attacks seek to change the behaviour of a victim DRL agent by adding an imperceptible perturbation to the agent's observations of the environment. As DRL agents are gaining success in tasks such as autonomous driving, there is a real potential for deploying these agents in safety-critical systems. Therefore, the threat of an adversary maliciously manipulating these agents' actions must be studied, as errors in safety-critical systems can have devastating consequences.

Behzadan and Munir [4], and Huang et al. [25] are the first to study the vulnerabilities of DRL agents to adversarial perturbation attacks. Since then, others have studied the

vulnerability of DRL agents under different settings. However, prior attack methods suffer from two major flaws. First, many proposed adversarial perturbation attacks [24, 54, 77] are very slow (they take longer than the time for the environment to move to the next observation), and require the adversary to pause the environment to send multiple versions of the current environment observation as inputs to the victim agent. These techniques require a powerful adversary that can control the environment of the DRL agent. Second, some attacks [25, 31] require the adversary to modify past observations stored in the victim agent's memory. This would assume a powerful adversary that has control over the victim agent's internal states. As these prior attacks against DRL agents require the adversary to have control of either or both the environment and the victim agent, it is unrealistic to deploy these attacks in real-time environments.

In this thesis, we propose new adversarial perturbation attacks and a new detection technique in real-time environments. We propose a more realistic adversary model where the adversary's capabilities are limited in the following two ways: 1) the adversary is constrained to apply adversarial perturbation to the current observation and cannot modify past observations stored in the victim's memory, and 2) the adversarial perturbation can only be computed and applied in a limited time before the victim collects the next observation in the environment.

Ownership Verification Model theft and unauthorized distribution of machine learning models are serious threats to model owners because training machine learning models is costly and resource-intensive. Ownership verification techniques such as watermarking and fingerprinting can effectively deter adversaries from stealing a model and they can be used to discover and prosecute adversaries for unauthorized use of stolen models.

There is extensive prior work on ownership verification of DNN classifier models through watermarking and fingerprinting. However, at the time that this thesis is written, there are only two techniques for ownership verification of DRL agents using watermarks [11, 3]. These two watermarking techniques for DRL agents require the verifier to either artificially generate environment observations or to passively wait for the suspect agent to interact with the environment until the watermarks are triggered. Both methods require the verifier to control the states generated by the environment. Additionally, watermarking require the watermarks to be embedded into the DRL agent's decision-making policy and require specialized training or retraining of the agent. Watermarking techniques are also susceptible to watermark removal techniques [13, 60].

In contrast to watermarking, fingerprinting utilize the intrinsic characteristic of an DRL agent and it works on any existing agents. We propose a novel fingerprinting method

that can verify the ownership of a suspect agent's policy, and the fingerprints are applied independently of the current observations of the environment.

Contributions In this thesis, we claim the following contributions:

- propose three new universal adversarial perturbations (UAPs): UAP-S, UAP-O, and OSFW(U), a modified version of Xiao et al.'s [77] attack (Section 4.1);
- provide an extensive evaluation of UAP attacks against three different DRL agents: DQN, A2C, and PPO. Using three different Atari 2600 games: Pong, Freeway, and Breakout (Section 6.1);
- propose a detection technique AD^3 that effectively detect UAP attacks (Section 4.2);
- provide an evaluation of AD^3 with respect to five different attacks: UAP-S, UAP-O, OSFW, OSFW(U), and FGSM (Section 6.2);
- propose a novel fingerprinting technique ReLF for ownership verification of DRL policies (Section 4.3);
- provide an evaluation of ReLF using three different DRL agents in Pong games (Section 6.3).

Thesis Structure The remainder of this thesis is organized into eight chapters. Chapter 2 contains background information required for understanding the materials presented in this thesis. Chapter 3 outlines the adversary models and requirements for each security topic. Chapter 4 presents designs of the proposed adversarial perturbation attacks, detection technique for adversarial perturbation attacks, and our proposed fingerprint technique for DRL policies. Chapter 5 details the environment, algorithm, and model architecture used for our experiments. Chapter 6 presents evaluations of the effectiveness of the proposed adversarial perturbation attacks and detection method, as well as a preliminary evaluation of the proposed fingerprint technique for DRL policies. Chapter 7 contains related work for both topics. Chapter 8 provides discussions on our proposed techniques and future works. Finally, Chapter 9 concludes this thesis with a summary.

Chapter 2

Background

This chapter presents background information that is vital to the understanding of this thesis. Section 2.1-2.3 focus on machine learning algorithms. Section 2.4 details the environments and the frameworks used in our experiments. Section 2.5 and 2.6 focus on prior adversarial perturbation attack techniques. Section 2.7 outlines existing solutions for ownership verification for deep neural network (DNN) classifier models.

2.1 Machine Learning

Machine learning is a discipline of computer science that studies algorithms that learn automatically through data [29]. There are three categories of machine learning techniques:

- Supervised learning: supervised learning methods use labeled training data of the form (x, y) where x is the feature vector, and y is the target class. The goal of supervised learning methods is to find associations between the input feature vector and the output class using the training data.
- Unsupervised learning: unsupervised learning methods use unlabeled training data. Unsupervised learning methods learn hidden patterns and relationships using unlabeled data.
- Reinforcement learning (RL): RL methods learn through an agent's interactions with the environment. The environment also consists of a reward signal that encourages reinforcement learning agents to learn desirable behaviours. RL methods find optimal decision-making policies of the agent by maximizing this reward.

2.2 Deep Neural Network (DNN)

Artificial neural networks (ANNs) are widely used in many machine learning applications and are inspired by human brains, mimicking the connections and signals between biological neurons. They are composed of hierarchical layers of interconnected neurons, consisting of an input layer, an output layer, and one or more hidden layers. A neuron is a processing unit that computes non-linear input-output mappings. Each neuron is connected to the neurons in the next layer and each connection has a weight and a threshold associated with it.

DNNs are ANNs with many hidden layers and can be described as parameterized functions $f(x, \theta)$. This parameter θ describes the weights of the neuron connections in the network. During training, different gradient-based optimization methods are used to find optimal values for θ.

For each input $x \in \mathcal{R}^n$ of n features, the function $f(x, \theta)$ outputs a vector $y \in \mathcal{R}^m$. The parameter vector θ is optimized by using a labeled training set D_{train}. For classification tasks, one can take a sample input $x \in \mathcal{R}^n$ and predict its label by taking $\hat{f} = \text{argmax}_i \, f_i(x, \theta)$ where each $i \in [1, ..., m]$ is the confidence score of x belonging to class i computed by f.

2.3 Deep Reinforcement Learning (DRL)

In RL, an agent interacts with the environment sequentially to learn a policy that maximizes an agent's *returns*: the accumulative rewards that the agent collects in a task. This problem can be formalized as a 5-tuple Markov Decision Process $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Where $\mathcal{S}$ is the state space that consists of multi-dimensional tensors; $\mathcal{A}$ is the action space consisting of possible actions that an agent can take, depending on the environment, an action is a vector of a fixed size and the values can be continuous or discrete; $\mathcal{P}$ is the transition kernel defining the dynamics of the environment that takes a state and an action and outputs the next state; $\mathcal{R}$ is the reward function that outputs a scalar value; $\gamma \in [0, 1]$ is the discount factor. This discount factor is used to balance the long-term and short-term impact of the agent's actions on the overall accumulative rewards.

RL tasks are sequential, the environment at each time-step t is characterized by a state $s_t \in \mathcal{S}$. For many environments, s_t does not consist of a single raw observation o. Instead, each raw observation o is preprocessed by a function f_{pre}, and often times, s_t consists of N past observations stored in memory; specifically, $s_t = \{f_{pre}(o_{t-N+1}), ..., f(o_t)\}$. Given a

state s_t at time t, an agent outputs an action a_t. With the agent performing action a_t, the environment transitions to the next time-step $t+1$ with a new state s_{t+1}, and $\mathcal{R}$ outputs a reward r_{t+1} based on the new state.

An agent has a policy π that is parameterized by θ and π maps states to action distributions $\pi(\cdot|s,\theta)$. This policy is used by the agent to decide on what actions to take at each state to maximize returns. RL techniques searches for a θ that maximizes the *expected discounted returns*: $G_t = \sum_{k \geq t+1} \gamma^{k-t} r_k$, which is the discounted accumulated rewards of an agent following π starting at time step t.

A value function outputs the expected discounted returns on a state for an agent following π. There are two types of value functions. The action-value function $Q^\pi(s,a;\theta) = E[G_t|s_t = s, a]$ is the expected discounted returns of action a in state s for an agent following π with parameter θ. Similarly, the value function of a state $V^\pi(s;\theta) = E[G_t|s_t = s]$ is the expected discounted return by an agent following π starting from state s.

There are different algorithms to approximate an agent's policy π. DRL algorithms utilize DNNs to approximate the policy π, the value function V, and/or the action-value function Q because DNNs are powerful function approximators.

There are two types of RL techniques, model-based and model-free techniques. Model-based techniques model both the environment dynamic $\mathcal{P}$ and the agent policy during training. This type of learning algorithm relies on expert knowledge of the environment and is beyond the scope of this thesis. Model-free techniques, on the other hand, estimate the optimal policy without modeling the environment dynamics and are used for this thesis.

Value-based Methods Instead of optimizing the policy directly, value-based methods optimize the value function. The policy is obtained by choosing the action that maximizes the expected returns at each state as defined by the value function. One example of such algorithm is Q-learning[45] that utilizes the Markov property of the RL formalization to define the action-value function through the Bellman equation: $Q(s_t, a_t) = G_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$. Then the action-value function is computed by minimizing the Bellman loss.

Policy-based Methods Policy based-methods directly parameterize the policy π and optimize its parameter θ to obtain an optimal policy. Typically, these methods optimize an objective function such as the expected discounted returns $E[G_t]$ through stochastic gradient ascent with respect to θ. An example of policy based-method is the REINFORCE algorithm[75] that updates the policy parameter θ using $\nabla \log \pi(a_t|s_t|\theta) G_t$ that is an estimator of $\nabla E[G_t]$.

Actor-critic Methods Actor-critic methods combine the approaches from both policy-based methods and value-based methods. The agent policy is split into two parts: an actor and a critic. The actor estimates the policy network and the critic estimates the value function of the current policy. The estimated value function is then used to assist gradient updates to the policy by reducing the gradient variance. An example of this method is A2C[43], which updates the policy parameter θ using $\nabla \log \pi(a_t|s_t|\theta)(G_t - V^{\pi}(s_t))$ that lowers the variance of the gradient.

Prior work on adversarial perturbation attack against DRL agents often use the following DRL algorithms to test their methods: Deep Q Network (DQN) [45], Trusted Region Optimization (TRPO) [58], Proximal Policy Optimization (PPO) [59], and Advantage Actor-Critic (A2C) [43].

2.4 DRL Testing Environments

While DRL agents have shown successes in complex tasks such as autonomous driving [52, 61] and robotic control [36], these complex tasks are not suitable for this thesis because 1) it is difficult to interface with the environments of these tasks and 2) DRL policies for complex tasks are difficult and time-consuming to train.

For testing our proposed methods, we use two environments from OpenoAI Gym[1]. OpenAI Gym provides a standardized interface for interacting with different environments commonly used for DRL research. This thesis utilizes the following two environments for evaluating our proposed methods. The first is the Atari 2600 Game Environment, where each task is an Atari game with varying complexity and the agent only has a small number of action choices. The second is MuJoCo Environment for Robotic Controls, where each task involves controlling a robot agent with varying numbers of limbs where the DRL agent's actions are in continuous spaces. We call these tasks *continuous control tasks* because the agent's actions are in continuous spaces.

2.4.1 Atari 2600 Game Environment

The Atari 2600 games are a collection of games that were released on the Atari 2600 gaming console. Bellemare et al. [6] developed the Arcade Learning Environment (ALE), a software framework that provides an interface to emulate the Atari 2600 games. A single

[1]https://gym.openai.com/

Atari game screen is composed of 160×210 pixels with a 128-colour palette and the frame rate of the Atari game emulator is 60 Hz. All possible video frames of an Atari game makes up the state space of the environment. Atari games are used for testing proof of concepts and provide a simple baseline for DRL algorithms, as the state space of Atari games is non-trivial (because it cannot be searched exhaustively), and the complexity of the games is limited due to the limitations of the hardware.

Pong (Figure 2.1a) Pong is a table tennis-like two-player game. The two players pass the ball back and forth by moving their paddles vertically up and down. A player scores a point if the opponent failed to pass the ball to the other side. In the Gym environment, the opponent (left peddle) is controlled by a black-box opponent with a fixed policy and the player controls the green paddle. An episode terminates when either the player or the opponent achieves 21 points in the game. The environment returns a reward when the player scores a point. The player is limited to six different actions: { up, down, noop, fire, shoot up, shoot down}.

Freeway (Figure 2.1b) Freeway is a game where the player controls a yellow chicken that starts at the bottom of the screen. The goal is to move the chicken across the screen to the other side of the freeway without getting hit by a car. The environment returns a reward when the player-controlled chicken crossed the road. An episode ends when the player reaches 34 points. The player is limited to three actions: {up, down, noop}.

Breakout (Figure 2.1c) In Breakout, there is a wall of bricks at the top of the screen. The player's goal is to control a paddle to hit the ball to break the bricks on the screen. The player begins the game with five balls, and a ball is lost when it hit the bottom of the screen. The game terminates when all balls are lost or when all bricks are removed. The environment returns a reward every time a brick is broken. The player is limited to six different actions: { left, right, noop, fire, shoot left, shoot right}.

2.4.2 MuJoCo Environment for Robotic Controls

MuJoCo[2] [69] which stands for Multi Joint dynamics with Contact is a physics engine used for robotics research. OpenAI Gym provides different robotic agents using the MuJoCo physics engine for basic tasks such as Cart-Pole balancing, and locomotive tasks such as

[2]https://mujoco.org/

(a) Pong (b) Freeway (c) Breakout

Figure 2.1: Screenshots of Atari 2600 games taken from the Arcade Learning Environment (ALE) using the Python Gym package.

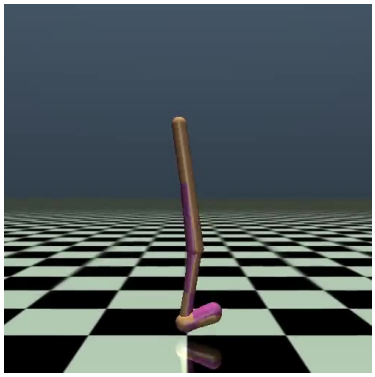
moving different robots (with a different number of limbs and center of mass) forward as fast as possible. The locomotive tasks are very challenging as the robotic agents have high degrees of freedom and only have a partial observation of the environment. Duan et al. [16] provide detailed descriptions and benchmarks for different RL policies on these locomotive tasks. Robotic control tasks are useful for evaluating DRL agents in tasks that are more challenging and complex than the Atari 2600 games.

2.5 Adversarial Examples

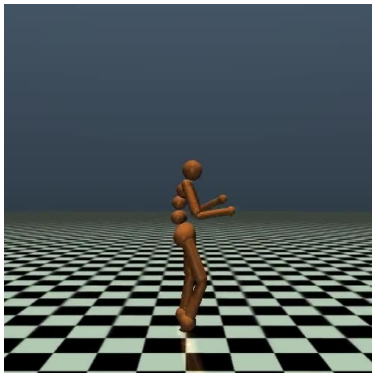
Szegedy et al. [67] were the first to discuss DNNs' vulnerability to adversarial examples. Given a valid input x and its corresponding class y, it is possible to find an adversarial example x^* that is close to x by some distance measure, but the class of x^* is not classified as y. When an adversary exploits this vulnerability by adding a perturbation to an input sample to fool a classifier, we call this an *adversarial perturbation attack*[3].

Given a classification model f and an input x, the problem of finding adversarial examples can be formulated as an optimization problem of adding an *adversarial perturbation* δ to x such that $x^* = x + \delta$:

[3]This is sometimes referred to in the literature as adversarial attack.



(a) Walker



(b) Humanoid

Figure 2.2: Images from OpenAI (https://gym.openai.com/envs/#mujoco) of two Robotic environments from MuJoCo. The goal is for the robot agent to move forward as fast as possible.

$$\underset{\delta}{\operatorname{argmin}} \|\delta\|_p \text{ s.t. } \underset{i}{\operatorname{argmax}}\, f_i(x+\delta) = t. \tag{2.1}$$

Where $t \neq \operatorname{argmax}_i f_i(x)$ for untargeted attacks and a single fixed t for targeted attacks. The size of the adversarial perturbation δ is constrained by an ϵ such that $\|\delta\|_p \leq \epsilon$ using the l_p norm. Solving this optimization problem is non-trivial because DNNs are highly non-linear and non-convex [34].

Goodfellow et al. [20] proposed the Fast Gradient Sign Method (FGSM) for generating untargeted adversarial examples. They theorize that although DNNs are non-linear, it is susceptible to linear adversarial perturbations. Therefore, they approximate a linear solution to Equation 2.1 by taking the sign of the gradient of the loss function l under the l_∞ norm:

$$\delta = \epsilon \cdot \operatorname{sign}(\nabla_x \ell(f(x,\theta), y)). \tag{2.2}$$

Adversarial examples can be computed very quickly using this method. However, FGSM does not return minimum adversarial perturbations. Kurakin et al. [33] built their idea based on FGSM and proposed an iterative version of FGSM that updates the adversarial perturbation δ using projected gradient descent (PGD) multiple times using Equation 2.2 with a learning rate. While this method is slower than FGSM, it solves the optimization problem more reliably and with a better solution. Papernot et al. [53] proposed a targeted adversarial example generation technique using the Jacobian Saliency Map Algorithm (JSMA). They utilize the gradient of the loss function to compute a saliency map,

which maps individual features in the input vector to its ability to direct the classification result to a specific class. The adversary picks several pixels of the input to modify until the class of the input vector changes. Carlini and Wagner [10] proposed three different targeted adversarial example generation techniques each targeting a different l_p norm. Baluja and Fischer [2] proposed a method that uses the Adversarial Transformer Network (ATN) to generate adversarial examples against targeted networks.

Moosavi-Dezfooli et al. [46] show the existence of universal adversarial perturbations (UAPs) that can be applied to different images and cause these images to be misclassified with high probability. They utilize the DeepFool algorithm [47] for finding UAPs.

2.6 Untargeted Adversarial Perturbation Attack in DRL

Prior work shows that DRL agents are also vulnerable to adversarial perturbation attacks. Untargeted adversarial perturbations apply adversarial perturbations to the victim's observations of the environment to mislead victim policies into taking sub-optimal actions.

Huang et al. [25] and Kos and Song [31] are the first to demonstrate that adversarial perturbation attacks are effective against DRL agents using FGSM attacks. Pattanik et al. [54] and Huai et al. [24] utilize PGD to find adversarial perturbations for different objective functions.

Xiao et al. [77] propose a technique ("obs-fgsm-wb", OSFW) that generates a single adversarial perturbation in the middle of an episode. The adversary collects the first k observations in the episode and calculates an adversarial perturbation using the average of the gradients over the k observations collected. This adversarial perturbation is then applied to all the subsequent observations of the episode.

While these untargeted attacks are effective in decreasing a victim agent's return in different tasks, they are difficult to deploy in practice. Many attacks [54, 24] are very slow because they require back-propagation to solve optimization problems. This is computationally intensive and requires sending the victim agent multiple versions of the current observation. Similarly, Xiao et al.'s [77] method requires pausing the environment mid-episode to compute an adversarial perturbation. These attacks require the adversary to have full control over the environment, which is unrealistic in complex real-time environments such as autonomous driving.

Other methods [25, 31] utilize FGSM in their attacks. FGSM requires collecting all observations in a state before computing the adversarial perturbation, which leads to the

adversary modifying past observations that are stored in the victim agent's memory. In this case, the adversary needs to have control over the victim agent's internal states, which is a strong assumption of the capability of the adversary.

The aforementioned attack techniques have an adversary with white-box knowledge of the victim agent such that the adversary has access to the weights of the policy's model(s) to generate adversarial perturbations. In some cases, it may not be possible for the adversary to gain white-box access to the victim's agent. Other prior attack methods construct their adversarial perturbations where the adversary only has black-box access to the victim agent. In this case, the adversary is often restricted to only having access to the action and the environment. We discuss black-box attacks in Section 7.1.2.

For this thesis, we focus on targeted attacks where the adversary has the singular goal of destroying the victim agent's performance. An adversary can have a more sophisticated goal of changing the specifics of the victim agent's behaviour and in this case, the adversary deploys an untargeted attack. However, as discussed in Section 7.1.1, these attacks are often very computationally heavy and are difficult to deploy in real-time.

2.7 Ownership Verification of DNNs

Machine learning models are increasingly deployed as a part of commercial applications. These models are valuable assets, as training machine learning models often requires a large number of computational resources and data. As a result, an adversary can gain profits by stealing the functionalities of a model and derive surrogate models to redistribute and monetize as their own.

A surrogate model is a copy of the source model that exhibits the same functionalities. Depending on the adversary model, there are different ways that this surrogate model is created. An adversary with white-box knowledge of the source model can simply copy the weights to the surrogate model. On the other hand, an adversary with black-box knowledge of the model needs to mount a model extraction attack using a set of unlabelled data and the source model's interface to train a surrogate model that emulates the functionalities of the source model.

In the case where theft cannot be prevented, model owners can claim ownership of a stolen model using ownership verification techniques. There are two major techniques that are discussed in the machine learning literature: watermarking and fingerprinting. For a comprehensive view of ownership verification techniques for DNNs, Franziska Boenisch [7] has a survey and analysis on this topic.

Watermarking Watermarking is commonly used in digital media where the owner's identification is infused with the media content such that its ownership can be verified after the content is distributed. A survey of watermarking techniques in digital media space can be found in Saini et al.'s survey [57].

This concept of watermarking the distributed contents can also be applied to machine learning models. The first watermarking technique for DNN models embeds a watermark to the model parameters. Song et al. [64] propose different methods to encode information into the least significant bit of the model parameters. Uchida et al. [72] embed the watermark to the convolutional layers of the DNN. This watermark is a t-bit binary vector. Their method modifies the original loss function used in training and adds an embedding regularizer that imposes a statistical bias on certain model parameters to embed this t-bit binary vector as watermark.

Other watermarking techniques utilize a set of special inputs to identify watermarked models instead of embedding the watermark to the model's parameters. This special set of inputs is often called the trigger dataset or carriers. As DNNs are known to be over-parameterized and are capable of memorizing random noisy inputs, it is possible to define a trigger dataset with limited harm to the utility of the model. Model owners would train the model to output specific target classes for the trigger set. During the verification stage, the model owner can verify the ownership of a suspect model by comparing the class agreement between the suspect model and the owner's model on the trigger dataset. Multiple works [1, 28, 35, 50] propose watermarking using trigger datasets that are generated from a distribution that is different from the training dataset. Another body of work [21, 37, 82] builds the trigger dataset by embedding information onto the data that is in the training set.

Notably, the same watermarking techniques cannot be directly applied to DRL policies. Watermarking techniques require modifications to the training of the models, however, the training algorithms and training environments of DRL policies are different from the training of DNN classifiers. Thus, the watermarking techniques used in DNN cannot directly transfer to the DRL case.

Fingerprinting Instead of embedding watermarks into DNNs, another approach is to find existing features of the model to uniquely identify stolen models. We call these unique identifiers fingerprints. There are two works [40, 83] on utilizing adversarial examples as fingerprints for ownership verification of DNNs models.

Lukas et al. [40] defined a new class of adversarial examples: conferrable adversarial examples, which is a type of adversarial example that is highly transferable to surrogate

models of a source model but not transferable to independently trained models. The verification algorithm can distinguish between independently trained models and surrogate models by testing the transferability of the conferrable adversarial examples. They proposed an ensemble method of generating conferrable adversarial examples by first generating a set of surrogate models and a set of independently trained models, then they utilize gradient optimization methods to optimize the conferrability score of a sample. This method is very costly for the defender because the defender has to generate many independent models to find these conferrable adversarial examples.

Zhao et al. [83] utilize a similar type of adversarial example that they defined as "adversarial marks". This type of adversarial example shows high transferability for similar models and not others. They utilize the targeted adversarial example generation technique proposed by Carlini and Wagner [10] with modified objective function and constraints. The authors argue that adversarial marks are difficult to remove as the number of possible adversarial examples is infinite.

These two fingerprinting techniques cannot be directly applied to DRL policies. They use a set of inputs to generate these adversarial examples and during verification, the same inputs are used to fingerprint a model. However, in DRL, the environments are not necessarily fixed and the verifier needs a strategy for applying adversarial perturbations to the appropriate states of the environment to fingerprint the model.

Chapter 3

Problem Statement

This chapter outlines the problem statement, adversary model, and design requirements for real-time adversarial perturbation attacks (Section 3.1), detecting adversarial perturbation attacks (Section 3.2), and ownership verification using fingerprinting (Section 3.3).

3.1 Real-time Adversarial Perturbation Attack

In adversarial perturbation attacks, we have deployed a deep reinforcement learning (DRL) victim agent v that interacts with the environment. As shown in Figure 3.1, the adversary Adv, sits between v and the environment and is able to add a bounded adversarial perturbation to v's observations of the environment.

3.1.1 Problem Statement

In this setting, we have a deployed DRL victim agent v that interacts with the environment and an adversary Adv that seeks to reduce v's returns in a task. We focus on the scenario where Adv compromises v's sensors and injects noise to v's observations of the environment. As shown in Figure 3.1, Adv sits between v and the environment and can add a bounded adversarial perturbation to v's observations of the environment. While prior work [25, 31, 54, 24, 77] demonstrated that DRL agents are vulnerable to adversarial perturbation attacks, these attack methods cannot be realized in real-time.

In this thesis, we study effective untargeted white-box adversarial perturbation attacks against DRL agents that can be realized in real-time.

3.1.2 Adversary Model

Adversary Goals In this setting, *Adv* mounts an adversarial perturbation attack against v in real-time where during attack deployment, *Adv* cannot pause or slow down the environment. This attack is an untargeted attack, where the goal is to lead v into taking sub-optimal actions and fail at its task during deployment. While *Adv* is mounting the attack against v, *Adv* also aims to be stealthy to evade detection and mitigation strategies deployed by the victim.

Adversary Capabilities *Adv* has the following capabilities:

1. has white-box access to v and its policy π_v, including the algorithm and model weights. Depending on the DRL algorithm, π_v is either the action-value function Q_v or the policy function π_v and the value function V_v;
2. can observe v for a small number of episodes and collect environment states and v's actions in these episodes;
3. can intercept v's observations of the environment by adding adversarial perturbations that are bounded by ϵ;
4. can only modify the observations sequentially and they cannot modify past observations when it is stored in v's internal memory.

In this setting, *Adv* does not have the ability to modify the environment or v. Instead, *Adv* intercepts the communication between the environment and v, i.e., v's observations of the environment. Capability #4 is defined with respect to this setting because the *Adv* cannot modify v's internal setup.

While *Adv* can modify v's observations of the environment, it cannot make drastic changes to these observations. If *Adv* makes drastic changes to v's observations of the environment, e.g., changing each observation to a blank screen, it would be easily detectable by v. It is also possible for v to deploy anomaly detection mechanisms to detect these changes to each observation to detect the presence of *Adv*. We define capability #3 to bound the size of the adversarial perturbation to avoid detection.

3.1.3 Attack Requirements

A successful real-time attack should satisfy the following requirements:

AR-1 can reduce v's returns on a task significantly and ideally by 100%;

AR-2 be mountable in real-time, where the time to compute and apply the adversarial perturbation combined with the agent response time is faster than the speed that the environment moves on to the next observation;

AR-3 can evade known detection and mitigation mechanisms such that the attack still retains the same effect on v.

We restrict the evaluation of requirement **AR-2** to only the within episode computation cost, where the attack should not interrupt and slow down the episode of the environment that is under attack.

3.2 Detecting Adversarial Perturbation Attacks

3.2.1 Problem Statement

As adversarial perturbation attacks can cause seriously failure to DRL agents, it is important to study techniques that can mitigate such attacks. Effective detection techniques can deter adversarial perturbation attack against the victim agent v. The detection mechanism is deployed as part of v and it alerts v when it senses the presence of Adv in the environment.

3.2.2 Adversary Model

The adversary model is identical to the adversary model in Section 3.1.2.

3.2.3 Detection Requirements

The requirements for a effective detection technique is outlined as follows:

DR-1 should minimize false positives and maximize true positives, i.e., can effectively raise an alarm in episodes where attacks are applied;

DR-2 should raise an alarm as early as possible when an attack is detected in an episode.

3.3 Ownership Verification Using Fingerprinting

3.3.1 Problem Statement

In this setting, there is a source policy π_o that is owned by a legitimate party. The adversary *Adv* gains access to π_o and creates a surrogate policy that is a copy of π_o. Then, *Adv* deploys an agent in an environment that has this surrogate policy. DRL agents that have π_o and its surrogate policies exhibit similar if not identical behaviours on a task.

The verifier *Ver* takes a suspect agent *sp* and seeks to verify whether or not it has a surrogate policy of π_o. As shown in Figure 3.1, during the verification stage, *Ver* sits between the environment and *sp* and adds a bounded adversarial perturbation to *sp*'s observation of the environment.

3.3.2 Adversary Model

Adversary Goals In this setting, *Adv* first creates a surrogate policy π_{Adv} that is a copy of π_o. *Adv* then deploy their own agent with π_{Adv}. *Adv* seeks to deploy their agent such that it has comparable performance to an agent with π_o. *Adv* also seeks to evade ownership verification that identifies π_{Adv} as a copy of π_o.

Adversary Capabilities *Adv* knows the fingerprinting techniques deployed and can modify the behaviour of the deployed policy to thwart any ownership verification attempts.

3.3.3 Verifier Model

Verifier Goals Given a source policy π_o and a suspect agent *sp* with policy π_s, *Ver* seeks to identify π_s to determine with high confidence whether or not π_s is a copy of π_o.

Verifier's Capabilities *Ver* has white-box access to π_o. Additionally, *Ver* has the following capabilities when interacting with *sp*:

1. has black-box access to *sp*, i.e., *Ver* does not know the algorithm and the policy weights of π_s;
2. can intercept *sp*'s observations of the environment by adding adversarial perturbations to them.

3.3.4 Verifier Requirements

To accomplish *Ver*'s goals, the fingerprint method needs to satisfy the following two requirements:

VR-1 should minimize false positives and maximize true positives, i.e., to not falsely verify the ownership of independently trained policies and to effectively verify ownership of policies that are copies of π_o;

VR-2 cannot be evaded by *Adv*.

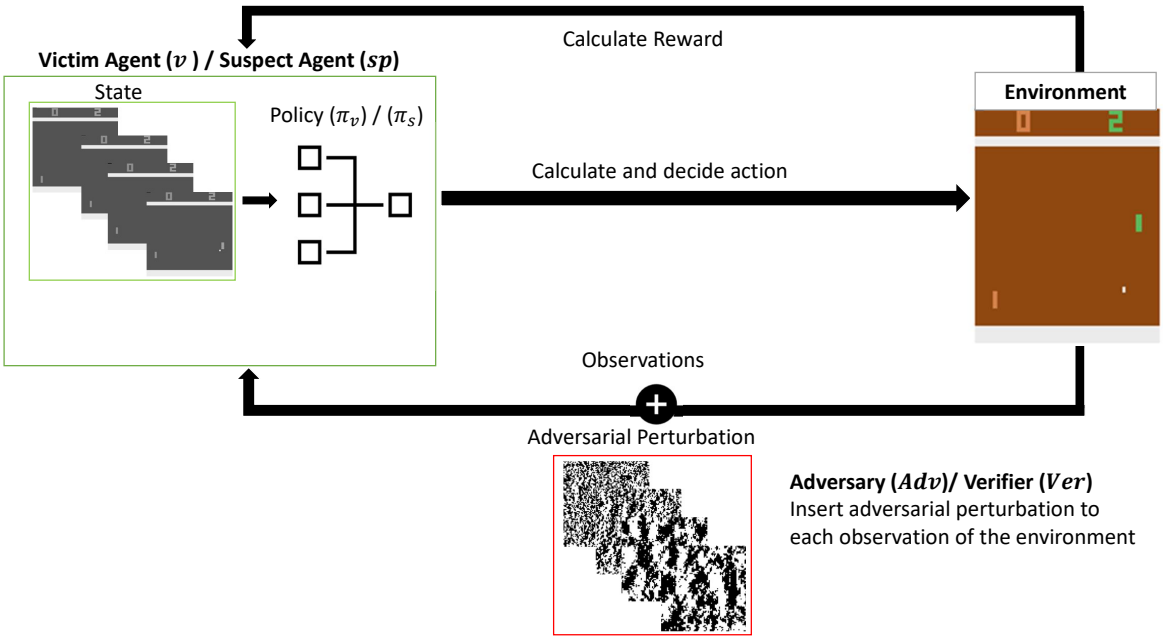


Figure 3.1: Interactions between the victim/suspect agent, the environment, and the verifier/adversary. The adversary/verifier adds a perturbation mask to victim/suspect agent's observations of the environment.

Chapter 4

Methodology

In this chapter, we present our proposed methods with respect to the problems outlined in Chapter 3. Section 4.1 presents three new real-time adversarial perturbations attacks (UAP-S, UAP-O, and OSFW(U)) against deep reinforcement learning (DRL) agents. Section 4.2 presents a new detection techniques for DRL agents against adversarial perturbations. Finally, Section 4.3 presents a novel fingerprint method to verify the ownership of stolen policies

4.1 Real-time Adversarial Perturbation Attack

In this section we discuss our attack designs to fulfill the requirements outlined in Section 3.1.3. We formally define three different universal adversarial perturbations (UAPs) against DRL agents. Our attacks are extended to the continuous control setting where agent actions are in continuous spaces. Finally, we discuss existing defense methods in DRL.

The methodologies detailed in this section of the thesis are also part of our Arxiv paper [68].

4.1.1 Attack Design

As discussed in Section 3.1.3, there is a set of requirements that the attacks should fulfill. We will address our design decisions in relation to each requirement.

To satisfy **AR-1**, we design attacks that lead the victim agent v into taking sub-optimal actions consistently because it can lead to a significant reduction to v's returns in a task. Moosavi et al. [46] propose UAP that generates a single perturbation that can be applied to any input data to a classifier and consistently mislead a classifier into predicting the wrong class for many of the given inputs. UAP is ideal for our purpose of consistently misleading v's actions. Based on their work, we propose two variants of UAP, state-agnostic UAP (UAP-S) and observation-agnostic UAP (UAP-O). A state s may consists of N processed observations $\{o_1, ..., o_N\}$. UAP-S generates an adversarial perturbation that is applied uniformly across different states but the perturbation is not uniform within the observations within a state. On the other hand, UAP-O generates an adversarial perturbation that is applied uniformly across all observations.

As discussed in Section 2.6, prior attacks are either too slow to satisfy requirement **AR-2** or require the adversary *Adv* to modify past observations in v's memory. To satisfy requirement **AR-2**, our attack methods pre-compute adversarial perturbations offline such that during deployment *Adv* only spends a minuscule amount of time to apply adversarial perturbations to v's observations of the environment. These adversarial perturbations are applied independently of the current state so that it does not require modifying past observations in v memory and only modify the most current observation. Xiao et al. [77] propose a similar technique OSFW that generates a single adversarial perturbation to reduce v's returns in a task. However, their method requires generating a new adversarial perturbation for each episode of a task and pausing in the middle of the task to compute this adversarial perturbation. This violates requirement **AR-2**. To facilitate fair comparison with this work, we design OSFW(U), that is a modified version of OSFW where a single adversarial perturbation is generated offline and can be applied in real-time.

4.1.2 Attack Implementation

The three attack methods UAP-S, UAP-O, and OSFW(U) pre-compute the adversarial perturbation using a training set D_{train}. This D_{train} goes through two phases, the first is the data collection phase and the second is the data sanitization phase. The quality of training data influences the effectiveness of the attack, as such, we sanitize D_{train} to improve the quality of D_{train}.

Attack Training Data Collection First, *Adv* copies the weights of the victim v's policy networks to a proxy network *adv*. Specifically, for value-based methods (e.g., DQN), weights of Q_v is copied into Q_{adv}, and for policy-based methods (e.g., A2C and PPO) both

the policy function π_v and the value function V_v are copied to π_{adv} and V_{adv}, respectively. We will refer to these networks as v's policy π_v. In addition, $Q_{adv}(s, a)$ can be computed from $V_{adv}(s)$ for each $a \in \mathcal{A}$. To simplify the notation used, we will use $\hat{Q}(s)$ to represent the action decision of a policy at state s.

Finally, while *Adv* is copying the network weights, *Adv* also observes and collects a set of states D_{train} by observing v's actions in an environment for one episode and collects all the states s in the episode.

Data Sanitization To better satisfy requirement **AR-1**, we improve D_{train} so that the adversarial perturbation r can be more effective in reducing the v's returns in a task. To improve D_{train}, *Adv* sanitizes D_{train} to contain only *critical states*. Following the definition from [38], critical states are the states in an episode where v's actions can heavily impact the returns in that episode. As a result of this, we want r to be the most effective in these states. In these states, v heavily prefers optimal actions over any other actions. We identify these critical states $s \in D_{train}$ using the action-preference function:

$$\begin{aligned} &\text{Var}_{a \in \mathcal{A}} \left[\text{Softmax}(Q_{adv}(s, a))\right] \geq \beta \\ \beta = \frac{1}{|\mathcal{D}_{train}|} &\sum_{s \in \mathcal{D}_{train}} \text{Var}_{a \in \mathcal{A}} \left[\text{Softmax}(Q_{adv}(s, a))\right], \end{aligned} \tag{4.1}$$

modified from Lin et. al. 's work [38]. Where Var is the normalized variance of $Q(s, a)$ for each $a \in \mathcal{A}$.

Generating UAP-S and UAP-O To generate UAP-S and UAP-O, *Adv* utilizes the states $s \in D_{train}$, by finding a perturbation r that satisfies Equation 2.1 for a *fooling rate* of δ on D_{train}. This fooling rate refers to the percentage of states s in D_{train} that r can successfully lead v into taking sub-optimal actions.

The implementation of UAP-S and UAP-O are modified versions of universal adversarial perturbations by Moosavi et al. [46] and relies on the Deepfool algorithm [47] to find minimal r that solves Equation 2.1 for $s \in D_{train}$. Both UAP-S and UAP-O seek to find a perturbation r such that $\hat{Q}_v(s + r) \neq \hat{Q}_v(s)$ for most state s that lead to v choosing the sub-optimal actions while performing a task. UAP-S finds r that is not uniform across all observations in a state. In contrast, UAP-O finds r such that the perturbation is uniform across all observations.

Algorithm 1 summarizes the method to generate UAP-S and UAP-O. To find minimal Δr in lines 5-6 of Algorithm 1, we utilize Deepfool to iteratively update perturbed state

$s_i^* = s + r + \Delta r_i$ at each iteration i until $\hat{Q}_{adv}(s_i^*) \neq \hat{Q}_{adv}(s)$. At each iteration i, DeepFool finds the closest hyperplane $\hat{l}(s_i^*)$ and Δr_i that projects s_i^* on the hyperplane. To find Δr_i, first, let $a_m = \hat{Q}_{adv}(s + r)$, for each $a_{\hat{l}} \in \mathcal{A}$ and $a_{\hat{l}} \neq a_m$, we compute Δr_i:

$$\begin{aligned} Q^{'}(s_i^*, a_{\hat{l}}) &\leftarrow Q_{adv}(s_i^*, a_{\hat{l}}) - Q_{adv}(s_i^*, a_m), \\ w_{\hat{l}}^{'} &\leftarrow \nabla Q_{adv}(s_i^*, a_{\hat{l}}) - \nabla Q_{adv}(s_i^*, a_m), \\ \Delta r_i^{'} &\leftarrow \frac{|Q^{'}(s_i^*, a_{\hat{l}})|}{\|w_{\hat{l}}^{'}\|_2^2} w_{\hat{l}}^{'} \end{aligned} \tag{4.2}$$

Where ∇ is the gradient of Q_{adv} w.r.t. s_i. Then we take Δr_i to be the smallest $\Delta r_i^{'}$ computed.

For UAP-S, the perturbation r is not uniform across the observations in a state $s_t = \{o_{t-N+1}, ..., o_t\}$. This means that $r = \{r_1, ..., r_N\}, r_i \neq r_j, \forall i \neq j$, and each r_i is applied to the corresponding observation $o_{t-N-i+1}$ within s when an attack is launched.

In contrast, for UAP-O, the perturbation is uniform across the observations in s_t. To construct UAP-O, we seeks to find a modified version $\hat{r}$ of perturbation r such that:

$$\begin{aligned} &\min(\|r - \hat{r}\|_2^2) \\ &\text{s.t.: } \hat{r}_j = \hat{r}_k, \forall j, k \in \{1, \cdots, N\} \text{ and } \|\hat{r}\|_\infty \leq \epsilon. \end{aligned} \tag{4.3}$$

The closest $\hat{r}$ to r can be found by modifying line 5 and 6 of Algorithm 1 that searches for Δr. The modified version searches for the closest $\Delta \hat{r}_i$ to Δr_i by averaging $w_{\hat{l}}^{'}$ over observations:

$$\Delta r_{ij}^{'} \leftarrow \frac{|Q^{'}(s_i^*, a_{\hat{l}})|}{N\|w_{\hat{l}}^{'}\|_2^2} \sum_{k=1}^{t} w_{\hat{l}k}^{'}, \tag{4.4}$$

In UAP-O, DeepFool algorithm returns $\Delta \hat{r}_i = \Delta \hat{r}_{ij}$ as the optimal perturbation. UAP-O adds the same perturbation $\hat{r}$ to every observation. In the environment where there is only a single observation in a state, i.e., $N = 1$, UAP-S is equivalent to UAP-O.

Algorithm 1: Computation of UAP-S and UAP-O

input : sanitized $\mathcal{D}_{train}, Q_{adv}$, desired fooling rate δ_{th}, max. number of iterations it_{max}, pert. constraint ϵ

output: universal perturbation r

Initialize $r \longleftarrow 0$, $it \longleftarrow 0$;
while $\delta < \delta_{max}$ ***and*** $it < it_{max}$ **do**
  for $s \in D_{train}$ **do**
    if $\hat{Q}(s + r) = \hat{Q}(s)$ **then**
      Find minimal Δr:
      $\Delta r \longleftarrow \text{argmin}_{\Delta r} \|\Delta r\|_2$ s.t. $\hat{Q}(s + r + \Delta r) \neq \hat{Q}(s)$;
      $r \longleftarrow \text{sign}(\min(\text{abs}(r + \Delta r), \epsilon))$;
  Calculate δ with the updated r on D_{train};
  $it \longleftarrow it + 1$;

Generating OSFW(U) As discussed in Section 2.5, OSFW requires pausing after collecting the first k states. It computes an adversarial perturbation r by averaging the gradient of Q_v, and applying it to the subsequent states in the episode. This method requires, 1) generating different r for each episode and 2) pausing in the middle of an episode to compute r. As calculating r in the middle of an episode requires *Adv* to pause the episode and thus conflict with requirement **AR-2**, we adapted OSFW and create a universal version OSFW(U) where we generate r offline. We utilize the proxy policy network Q_{adv} and take the first k states from $\mathcal{D}_{train}$ that were collected. The formula to calculate the adversarial perturbation r for OSFW(U) is:

$$r = \epsilon \cdot \text{sign}\Big(\frac{1}{k}\sum_{i=0}^{k} \nabla_{s_i}(-\log(Q_{adv}(s_i, \hat{a}_i)))\Big); \tag{4.5}$$

Where $\hat{a}_i = \hat{Q}_{adv}(s_i)$ is the chosen action at state $s_i \in D_{train}$.

4.1.3 Extending Attack to the Continuous Control Setting

In continuous control environments, v's selected action is an array of real values. The DRL agent in this setting controls physical systems such as humanoid and other multi-joints robots in this environment [69]. There is no action-value function $Q(s, a)$ in this

Algorithm 2: Computation of UAP-S and UAP-O in the continuous control setting

input : sanitized $\mathcal{D}_{train}, V_{adv}$, hyper-parameter α,
max. number of iterations it_{max}, pert. constraint ϵ
output: universal perturbation r

1 Initialize $r \longleftarrow 0$, $it \longleftarrow 0$;
2 **while** $\delta < \delta_{max}$ ***and*** $it < it_{max}$ **do**
3 **for** $s \in D_{train}$ **do**
4 **if** $V_{adv}(s+r) + \alpha \geq V_{adv}(s)$ **then**
5 Find the minimal Δr:
6 $\Delta r \longleftarrow \text{argmin}_{\Delta r} \|\Delta r\|_2$ s.t. $V_{adv}(s+r+\Delta r) + \alpha < V_{adv}(s)$;
 $r \longleftarrow \text{sign}(\min(\text{abs}(r+\Delta r), \epsilon))$;
7 Calculate δ with the updated r on D_{train};
8 $it \longleftarrow it + 1$;

setting because the action space is continuous. Nonetheless, *Adv* still has access to the value function $V(s)$. This function is an approximation of the expected discounted return of a state s by an agent with policy π. *Adv* can find an adversarial perturbation r that lowers the value of a state s. Because $V(s+r)$ is lower than $V(s)$, it can lead v into choosing a less optimal action. For FGSM, OSFW, and OSFW(U), *Adv* can use the gradient of V_{adv} instead of Q_{adv}. For UAP-S and UAP-O, lines 4 - 6 of Algorithm 1 is modified to only use V_{adv}. Algorithm 2 is corrected for generating UAP-S and UAP-O for continuous control environments.

4.1.4 Existing Defenses in DRL

To evaluate our attacks with respect to requirement **AR-3**, we present existing mitigation methods for DRL agents against adversarial perturbation attacks.

Adversarial retraining [5, 31] has shown to be a promising defense against adversarial perturbation attacks in DRL. However, adversarial retraining often leads to unstable training and performance degradation. Moreover, Moosavi et al. [46] show that while UAP-Ss have lower fooling rate on the test set compared to the training set, *Adv* can generate unlimited number of UAP-S in which adversarial retraining cannot defend against. Therefore, adversarial retraining techniques cannot be applied in this setting directly.

Zhang et al. [81] propose state-adversarial Markov decision process (SA-MDP), a method for adversarial retraining of DRL policies that also maintains v's performance. This method aims to find optimal policy π by training against *Adv* through policy regularization and maintaining the top-1 action of the policy.

Another defense technique is Visual Foresight. Lin et al. [39] propose Visual Foresight as a detection and recovery technique that utilizes an action-conditioned frame prediction module. The architecture of this prediction module is based on the architecture proposed by Oh et al. [51]. At every time-step t, the action-conditioned frame prediction module take the first k previous observations $o_{t-k} : o_{t-1}$ and k previous actions $a_{t-k} : a_{t-1}$ to predict the observation $\hat{o}_t$ at time t. Utilizing the partially predicted state $s_{pred} = \{o_{t-N+1}, ..., \hat{o}_t\}$ and the actual current state s_t, we can compare the distance between the action distribution of $Q(s_t)$ and $Q(s_{pred})$. In this work, they utilize l_1 as the distance metric $D(\cdot, \cdot)$ to compare the action distribution. In detection mode, if $D(Q(s_t), Q(s_{pred}))$ exceeds a predefined threshold H, then an attack is detected; to recover policy performance, $Q(s_{pred})$ can be used when an attack is detected at each state. However, they did not propose a method for selecting this threshold H.

4.2 Detecting Adversarial Perturbation Attacks

This section presents our detection technique AD^3. Section 4.2.1 outlines the design choices for AD^3 and Section 4.2.2 details the implementation of AD^3.

4.2.1 Detector Design

While mitigation methods can be used for recovering the performance of v in many tasks, a detection technique can be more effectively deployed in tasks with a clear negative result from an episode (such as losing a game), or the possibility of negative returns. A reasonable response for v is to suspend the episode when *Adv* is detected to prevent the negative result. We propose a detection method, Action Distribution Divergence Detector (AD^3), for v to detect the presence of *Adv*.

In an episode of an DRL agent acting on a task, the sequence of actions exhibits some degree of *temporal coherence*, i.e, the likelihood of the agent choosing a specific action given a specific prior action is similar across different episodes. We observed that this temporal coherence is disturbed when the episode is under attack. We utilize this information to design AD^3 by calculating the statistical distance between the *conditional action probability*

distribution (CAPD) of the current episode to the learned CAPD of past episodes to detect attacks. To satisfy requirement **DR-1**, we compare the learned CAPD against episodes with no attack to find a suitable threshold where CAPD of normal episodes usually falls under and CAPD of episodes under attack would usually exceed.

Unlike prior work on detecting adversarial examples in the image domain [42, 56, 79], AD^3 does not analyze the input image nor try to detect adversarial examples. Instead, it observes the distribution of the actions triggered by the inputs and detects unusual action sequences.

4.2.2 AD^3 Implementation

To train AD^3, we first run k_1 episodes of v in a controlled environment before deploying AD^3. All actions taken during these episodes are recorded. These are actions used to approximate the conditional probability of the next action given a prior action using the bi-gram model. The conditional probability of actions approximated in these episodes isa called the *learned* CAPD.

To fulfill requirement **DR-1**, AD^3 needs to reliably differentiate between CAPD of a normal episode versus an episode that is under attack. To achieve this, AD^3 models the CAPD between normal episodes and the learned CAPD by running another k_2 episodes of v in a controlled environment. AD^3 calculates the statistical differences between the CAPD of the normal episodes and the learned CAPD at each time-step using Kullback-Leibler (KL) divergence [32]. A threshold value th is chosen such that the KL divergence between the two CAPDs can mostly fall under.

For each episode in the second run of k_2 episodes, the KL divergence between the learned CAPD and the CAPD of the current episode is removed for the first t_1 steps because the CAPD of the current episode is unstable and KL divergence is naturally high in the beginning. We set the threshold th as the p^{th} percentile of all KL divergence values calculated in this k_2 episodes.

During deployment, AD^3 continuously updates the CAPD of the current episode, and after t_1 time-steps, it calculates the KL divergence between the CAPD of the current episode and the learned CAPD. If the KL divergence exceeds the threshold th by $r\%$ or more during a time window t_2, in accordance with requirement **DR-2**, AD^3 immediately raises an alarm that the policy is under attack.

To illustrate how AD^3 works, we look at an example of a DQN agent playing Pong (from optimal parameters for AD^3 for the Pong DQN agent in Table 5.1), we use $k_1 = 12$,

$k_2 = 24$, and $p = 100$ to calculate threshold $th = 0.19$. We set $t_1 = 200$, $t_2 = 100$, and $r = 90$. In Figure 4.1, we show the KL divergence between the learned CAPD and the CAPD of a normal episode and an episode under FGSM attack. As shown in this figure, AD^3 does not raise an alarm during the first 200 steps, as the KL divergence for an episode under attack and a normal episode is high (and above th) for the first 200 steps. After 200 steps, AD^3 does not raise an alarm for a normal episode because the KL divergence at each time-step afterwards is consistently below th. In contrast, for the episode that is under attack, an alarm is raised at step 300 because 100% of the the KL divergence exceeds th for t_2 steps after t_1.

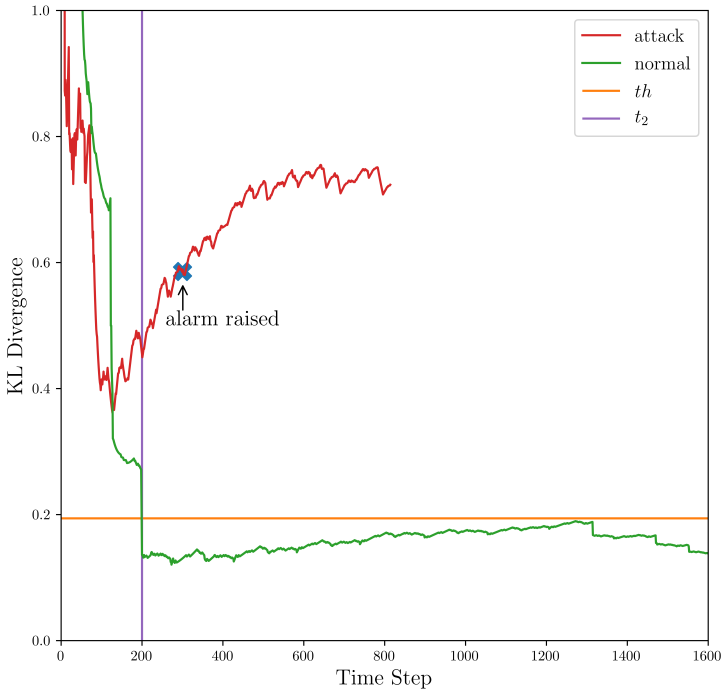


Figure 4.1: The KL divergence between the learned CAPD and the CAPD of two different episodes of a DQN agent playing Pong. One episode is a normal episode (in green) and the other is an episode under FGSM attack (in red). AD^3 is deployed in both episodes and the time step where an alarm is raised is labeled with a blue X marker.

4.3 Ownership Verification using Fingerprinting

We propose a novel fingerprinting method *Reinforcement Learning Fingerprinting* ReLF for ownership verification of DRL policies. Section 4.3.1 discusses the design choices of ReLF. Section 4.3.2 details the algorithm and the implementation of fingerprint generation and how it is used to verify the ownership of a suspect agent's policy.

4.3.1 Fingerprinting Design

The effectiveness of UAPs is directly dependent on the source policy's randomization during training and the source policy's actions collected in the training set. This constrains the transferability of UAPs across policies. As a result, two agents that have two independently trained policies would act differently in an episode where UAPs are applied because the effectiveness of UAPs differs between policies.

We measure the difference in actions between two policies by calculating *sample action agreement* (SAA) between the sequences of actions between two agents in a task. If the SAA between two agents is low in a task where UAPs are applied, then we can conclude that these two agents have two independently trained policies. We take random samples from an episode's action sequences to make it difficult for an adversary to change the actions in an episode that are used to verify the ownership of their agent.

We define SAA as the sampled average percentage of actions that are the same between two sequences of actions. The sample action agreement function $SAA(A_1, A_2, m, n)$ take a m random samples of n indices from A_1 and A_2 without replacement and compares the action agreement between the two sequences at the sampled indices. Algorithm 3 details the algorithm to calculate SAA.

The indicator function $\mathbb{I}$ used in Algorithm 3 is defined as:

$$\mathbb{I}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

From our preliminary experiments using Pong games, we found that in a normal episode with no adversarial perturbation attack, the SAA can be low between agents with the same policy. In a normal episode with no attacks, the SAA between the same A2C policy is around 0.2, and for PPO policies, the SAA between the same policy is around 0.6. Notably, the SAA between independently trained policies for A2C and PPO is 0.2 in normal episodes.

Algorithm 3: Calculate sample action agreement SAA

input : Action sequence $A_1 = (a_{11}, ..., a_{1_k})$, second action sequence $A_2 = (a_{2_1}, ..., a_{2_k})$, number of samples m, sample population size n

output: sample action agreement (SAA) p

Initialize $pArr \longrightarrow [\,]$

for $j \leftarrow 1$ *to* m **do**

  $I \longleftarrow$ a random sample of of n elements from $[1, 2, ..., k]$ without replacement

  $pArr[j] \longrightarrow \frac{1}{n} \sum_{i \in I} \mathbb{I}(a_{1_i}, a_{2_i})$

$p \longrightarrow Average(pArr)$

return p

Thus, the SAA is indistinguishable between independently trained policies and the same policy for A2C. While the SAA between the agents with the same PPO policy is more distinguishable between the same policy and independently trained policies, the difference in SAA can be even more prominent in an environment where the states are adversarially perturbed.

When we generate OSFW(U) against a source policy and apply it to an episode, the SAA between agents with the source policy and its copies can go up to as high as 0.9. Following this observation, high SAA between two agents in an episode where OSFW(U) is applied is an indication that the two agents have the same policies.

We also found that compared to UAP-S and UAP-O, OSFW(U) is the most effective in leading agents with independently trained policies to have low SAA (as low as 0.1 in some cases) between the agents. However, there are some instances where applying OSFW(U) in an episode leads to a high SAA (higher than 0.8) between two agents that follow two independently trained policies. To minimize false positives, we generate multiple adversarial perturbations using OSFW(U) and apply each of them to different episodes of a task. A policy is only identified as a copy of the source policy if the SAA is high for all episodes. Therefore, to satisfy requirement **VR-1**, we use OSFW(U) adversarial perturbations to reliably identify copies of the source policy, and we use multiple OSFW(U) adversarial perturbations to minimize false negatives of ReLF and increase the confidence of detecting stolen policies.

4.3.2 Fingerprinting Implementation

Given a source policy π_o and a suspect agent sp with a policy π_s, our fingerprinting method, ReLF seeks to identify ownership of π_s. ReLF consists of two algorithms. The first is **Generate**, that generates a set of secret fingerprints $\mathcal{F}_{\pi_o}$ of π_o; the second is **Verify**, that confirms or rejects π_s to be a copy of π_o using $\mathcal{F}_{\pi_o}$.

- **Generate**(π_o): Generates a set of fingerprints $\mathcal{F}_{\pi_o}$ of size num_f for π_o.
- **Verify**(sp, $\mathcal{F}_{\pi_o}$): Confirms or rejects π_s to be a copy of π_o using $\mathcal{F}_{\pi_o}$. The algorithm returns 1 if π_s's ownership is proven by $\mathcal{F}_{\pi_o}$, i.e., π_s is a copy of π_o; otherwise, the algorithm returns 0 and fails to prove the ownership of π_s, i.e., π_s is not a copy of π_o.

Fingerprint Generation Given a source policy π_o, a set of fingerprints $\mathcal{F}_{\pi_o}$ is a tuple of size num_f such that $\mathcal{F}_{\pi_o} = \{(r_1, p_1), ..., (r_{num_f}, p_{num_f})\}$. Each $(r_i, p_i) \in \mathcal{F}_{\pi_o}$ consist of two parts: r_i is a OSFW(U) perturbation mask; and p_i is the SAA between two action sequences, one from π_o and the other from a copy of π_o in an episode where r_i is applied. The parameters m and n are used to calculate SAA are predefined values.

We utilize a modified version of OSFW(U) to compute perturbation mask r_i. Instead of taking the first k states from D_{train} as in Section 4.1.1, we sample random k states from D_{train} to generate each OSFW(U) mask. This enables us to generate different OSFW(U) masks from the same training set.

Generate first collects a set of training data D_{train} by observing π_o acting in an environment for one episode and collects all states in the episode.

The steps to generate $\mathcal{F}_{\pi_o}$ is summarized in Algorithm 4. To generate each tuple $(r_i, p_i) \in \mathcal{F}_{\pi_o}$ **Generate** does as follows:

1. Generate an adversarial perturbation $\tilde{r}_i$ using the modified OSFW(U) on D_{train} that is constraint by ϵ.
2. Initialize an agent v with π_o and a policy $\hat{\pi}_v$ that is a copy of π_o.
3. Deploy v in an environment where $\tilde{r}_i$ is applied for one episode. The episode ends at time-step t_{max}. At each time-step t, v takes $\tilde{s}_t = s_t + \tilde{r}_i$ as input and outputs an action a_t to the environment. We record the entire action sequence in the episode as $A_v = (a_1, ..., a_{t_{max}})$. For cross-checking, the same state $\tilde{s}_t$ is feed into $\hat{\pi}_v$ at each

time-step t and its action output $\hat{a}_t$ is recorded and stored into an action sequence $\hat{A}_v = (\hat{a}_1, ..., \hat{a}_{t_{max}})$. Notably, only v with π_o is interacting with the environment. The action outputs of $\hat{\pi}_v$ are recorded but not acted upon by an agent.

4. Calculate $\tilde{p}_i = SAA(A_v, \hat{A}_v, m, n)$. While $\hat{\pi}_v$ is a copy of π_o with the same weights and algorithm, the output actions between the two policies can still differ for stochastic policies.

5. If $\tilde{p}_i > \tau_1$ for a predefined threshold τ_1, then we set $p_i = \tilde{p}_i$ and $r_i = \tilde{r}_i$. Otherwise, go back to step 1.

Algorithm 4: ReLF algorithm - **Generate**

input : source policy π_o, training dataset D_{train}, number of fingerprints to generate num_f, environment env, decision threshold τ_1, sample size m, sample population size n

output: A set of fingerprints $\mathcal{F}_{\pi_o}$

Initialize policy $\hat{\pi}_v$ that copies the weights and algorithm of π_o
Initialize $\mathcal{F}_{\pi_o} \longleftarrow \{\}$
while *size of* $\mathcal{F}_{\pi_o} \neq num_f$ **do**
    $adv \longleftarrow$ generate modified OSFW(U) adversarial perturbation using D_{train}
    $A_v \longleftarrow [\,]$
    $\hat{A}_v \longleftarrow [\,]$
    $s \longleftarrow env.reset()$
    $t \longleftarrow 0$
    while *current env episode not finished* **do**
        $\hat{s} \longleftarrow s + adv$
        $a \longleftarrow \pi_o.select(\hat{s})$
        $\hat{a} \longleftarrow \hat{\pi}_v.select(\hat{s})$
        $A_v[t] \longleftarrow a$
        $\hat{A}_v[t] \longleftarrow \hat{a}$
        $s \longleftarrow env.step(a)$
        $t \longleftarrow t + 1$
    $\tilde{p} \longleftarrow SAA(A_v, \hat{A}_v, m, n)$
    if $\tilde{p} > \tau_1$ **then**
        $\mathcal{F}_{\pi_o} \longleftarrow \mathcal{F}_{\pi_o} \cup (adv, \tilde{p})$
return $\mathcal{F}_{\pi_o}$

Fingerprint Verification Given a suspect agent *sp* with policy π_s and a set of fingerprints $\mathcal{F}_{\pi_o}$ of π_o. To verify the ownership of π_s, **Verify** adds each adversarial perturbation r_i in $\mathcal{F}_{\pi_o}$ to *sp*'s observations of the environment.

If *sp* shows high SAA with π_o for the majority (over 50%) of the episodes where each of the adversarial perturbation $r_i \in \mathcal{F}_{\pi_o}$ is applied, then π_s is confirmed to be a copy of π_o. **Verify** is summarized in Algorithm 5.

More formally, **Verify** verifies π_s through the following steps. For each tuple $(r_i, p_i) \in \mathcal{F}_{\pi_o}$, **Verify** does as follows:

1. Deploy *sp* in an environment where r_i is applied in an episode that ends at time t_{max}. At each time-step t, *sp* observes $\hat{s}_t = s_t + r_i$ as input and outputs an action a_t to the environment. The sequence of actions by *sp* is recorded as $A_s = (a_1, ..., a_{t_{max}})$. Simultaneously, inject the same state $\hat{s}_t$ to π_o and record the action $\hat{a}_t$ selected by π_o. We record this action sequence as $A_o = (\hat{a}_1, ..., \hat{a}_{t_{max}})$. Only *sp* with π_s interacts with the environment. The action outputs of π_o are recorded and not acted upon by an agent.
2. Calculate $\tilde{p} = SAA(A_s, A_o, m, n)$.
3. If $p_i - \tilde{p} \leq \tau_2$, confirm the ownership of π_s using (r_i, p_i). Else (r_i, p_i) rejects π_s to be a copy of π_o.

If the ownership of π_s is confirmed by the majority (over 50%) of the tuples $(r_i, p_i) \in \mathcal{F}_{\pi_o}$, then **Verify** returns 1 to confirm the ownership of π_s. Otherwise, **Verify** return 0 and fails to prove the ownership of π_s.

Algorithm 5: ReLF algorithm - **Verify**

input : source policy π_o, fingerprints $\mathcal{F}_{\pi_o}$, suspect agent sp with policy π_s, environment env, verify decision threshold τ_2, sample size m, sample population size n

output: 1 to confirm the ownership of π_s and 0 to reject and fails to prove the ownership of π_s

foreach $(r, p) \in \mathcal{F}_{\pi_o}$ **do**
    $A_v \longleftarrow [\,]$
    $A_s \longleftarrow [\,]$
    $decision \longleftarrow [\,]$
    $t \longleftarrow 0$
    $s \longleftarrow env.reset()$
    while *current env episode not finished* **do**
        $\hat{s} \longleftarrow s + r$
        $a \longleftarrow \pi_o.select(\hat{s})$
        $\hat{a} \longleftarrow sp.select(\hat{s})$
        $A_v[t] \longleftarrow a$
        $A_s[t] \longleftarrow \hat{a}$
        $s \longleftarrow env.step(\hat{a})$
        $t \longleftarrow t + 1$
    $\tilde{p} \longleftarrow SAA(A_v, A_s, m, n)$
    if $p - \tilde{p} \leq \tau_2$ **then**
        $decision.append(1)$
    else
        $decision.append(0)$
if $Average(decision) > 0.5$ **then**
    return 1
else
    return 0

Chapter 5

Experimental Setup

In this chapter, we outline the hardware and software used for all of our experiments. Section 5.1 outlines the hardware and software setup used. Section 5.2 outlines the setup for the Atari Environment. Section 5.3 outlines the parameters and environment setup for evaluating both the attack methods and prior defense methods against adversarial perturbation attacks. Section 5.4 outlines the parameters used for our detection technique AD^3. Finally, Section 5.5 outlines the environment setup and parameters of our fingerprint method ReLF for ownership verification experiments.

5.1 Software and Hardware Setup

For consistency, we utilize the same hardware and software setup for all of our experiments. All implementation and experiments are done using Python version 3.7, using packages: PyTorch (version 1.2.0), NumPy (version 1.19.1), Gym (version 0.15.7), and MuJoCo 1.5 libraries. The experiments were done on the following two machines:

- A computer with 16 core Intel(R) Core i9 Processors with 64 GB RAM, and with NVIDIA GeForce RTX 2080 Ti with 11 GB.
- A computer with 2×12 cores Intel(R) Xeon(R) CPUs with 32 GB RAM and NVIDIA Quadro P5000 with 16GB memory.

5.2 Atari Environment

All training and testing are done using OpenAI Gym through the Gym package in Python. We use three Atari 2600 games, Pong, Freeway, and Breakout games to evaluate our methods. The Atari 2600 games run at 60 Hz where the time interval between each frame is $1/60 = 0.017$ seconds. Each frame in the game is composed of 210×160 RGB pixels.

Each frame from the Atari environment is pre-processed such that each RGB frame is converted to gray-scale, the size of the original frames of 210×160 pixel are resized to 84×84, and each pixel value is normalized from $[0, 255]$ to $[0, 1]$. To include temporal information to the input of DRL policy, we stack N pre-processed frames together. The state s input to the DRL policy is composed of N pre-processed frames that we refer to as observations o in this thesis. A new action is selected for every state and this action is repeated until the next state. We choose $N = 4$ such that the resulting s is a vector of size $4 \times 84 \times 84$.

We utilize the same DNN architecture as in Mnih et al.'s work [45] for the action-value function Q_v in DQN and the value function V_v for A2C and PPO. The DNN consists of three convolutional layers. The first hidden layer consists of 32 convolutional kernels of the size of 8×8 with a stride of 2. The second hidden layer consists of 64 convolutional kernels of size 3×3 with a stride of 1. The third hidden layer is a fully-connected layer with the output of size 512. Finally, the output layer is a fully-connected layer that maps the 512 output to output for each valid action. For the games used for this thesis, the number of possible actions is either 3 (for Freeway) or 6 (for Pong and Breakout). Our implementation of DRL algorithms is based on OpenAI baselines[1], and our DRL agents achieve similar average returns.

5.3 Real-Time Adversarial Perturbation Attacks

5.3.1 Setup for Evaluating Adversarial Perturbation Attacks

For UAP-S and UAP-O we set the desire fooling rate δ_{max} to 95% so that the algorithm stops searching for adversarial perturbation r when $\delta \geq 95\%$. For OSFW(U), we utilize the first 60 states of D_{train} to generate r. As a baseline comparison, we implemented FGSM, and OSFW, and fixed random noise. We measure the attack efficiency for $\epsilon =$

[1] https://github.com/DLR-RM/rl-baselines3-zoo

$\{0.01, 0.02, ..., 0.1\}$ and report the average return over 10 episodes with a different seed between testing and training.

5.3.2 Setup for Prior Defense Techniques

To evaluate adversarial perturbation's robustness against existing defense methods, we evaluate the attacks against two prior defense methods SA-MDP and Visual Foresight.

For SA-MDP, we downloaded the trained state adversarial policy for DQN (SA-DQN) [2]. The SA-DQN policy is trained with adversarial noise constraints by $l_\infty = 1/255$. Notably, when we try to train SA-DQN with adversarial noise constraint by $l_\infty = 0.01$ and the policy cannot retain its performance. Their SA-DQN policy is trained using one observation per state. Therefore, UAP-S reduces to UAP-O when evaluating this method.

We implemented and trained Visual Foresight for DQN agents following their original experiment setup and setting $k = 3$ to predict every 4^{th} observation. As we are unable to train a good prediction module for Breakout, we exclude Breakout for our defense experiments.

5.3.3 Setup for Adversarial Perturbation Attack in Continuous Control

The only policy-based algorithm can be used in a continuous control environment as the action-value function Q is only applicable for discrete action space. Between PPO and A2C, we chose to evaluate our attack using PPO because it is more recent compared to A2C and it is considered to be the state-of-the-art in reinforcement learning. We utilize pre-trained PPO agents for two different MuJoCo tasks: Humanoid and Walker2d. The PPO agents are downloaded from the GitHub repository[3] by Zhang et al. [81]. Notably, each state in this training environment contains only a single observation. Therefore, UAP-S reduces to UAP-O against these two agents. For UAP-S(O), OSFW , and OSFW, we copied the weights of V_v to V_{adv} and utilize V_{adv} instead of the action value function Q_{adv}. Additionally, we generated UAP-S(O) using $\delta_{max} = 95\%$. We evaluate the effectiveness of the attacks for ϵ from 0.0 to 0.2.

[2] https://github.com/chenhongge/SA_DQN
[3] https://github.com/huanzhang12/SA_PPO

5.4 Detecting Adversarial Perturbation Attacks

5.4.1 Setup of Atari 2600 Breakout Games

Many episodes of DRL agent playing Breakout took a long time to run (until the environment reaches the default maximum time-step). As discussed in Section 2.4.1, an episode of Breakout game terminates only when all balls are lost or when all bricks are removed. This creates an issue where, if an agent can always catch the ball before it falls through the screen but the agent cannot clear all the bricks, the agent is stuck in an infinite loop that continues until the default maximum time-step is reached.

This infinite loop results into two issues: 1) the experiments takes a long time to run, and 2) the action sequences in the infinite loop take over the distribution of the trained conditional action probability distribution (CAPD) in AD^3. The second issue makes AD^3 ineffective in Breakout because CAPD is skewed heavily to model the action sequences in the infinite loop. To resolve the above issues, we modify the environment of Breakout and change the maximum time-step of the environment from 99999 to 3000. At time-step 3000, a DRL agent has already cleared as many bricks as it can and cannot increase its returns anymore.

5.4.2 Setup for AD^3

As there are many parameters in AD^3, we first perform a grid search to find optimal parameters for AD^3 in Breakout, Pong, and Freeway using three different agents (DQN, A2C, and PPO) and choose the parameters with the best F1-score against five different attacks at $\epsilon = 0.01$. The optimal parameter values can be found in Table 5.1.

We then measure the false positive rate and false negative rate of AD^3. To calculate the false positive rate of AD^3, we report the fraction of episodes with alarms raised over 10 episodes. Similarly, to calculate the true positive rate of AD^3, we mount each attack at $\epsilon = 0.01$ and report the fraction of episodes with alarms raised over 10 episodes. We use different seeds for training AD^3, finding optimal parameters for AD^3, and for testing AD^3.

Game	DQN Parameters	A2C Parameters	PPO Parameters
Pong	$k_1 = 12$ $k_2 = 24$ $p = 100$ $r = 90$ $t_1 = 400$ $t_2 = 200$	$k_1 = 12$ $k_2 = 24$ $p = 95$ $r = 90$ $t_1 = 400$ $t_2 = 100$	$k_1 = 12$ $k_2 = 24$ $p = 100$ $r = 90$ $t_1 = 400$ $t_2 = 200$
Freeway	$k_1 = 12$ $k_2 = 24$ $p = 90$ $r = 80$ $t_1 = 300$ $t_2 = 200$	$k_1 = 12$ $k_2 = 24$ $p = 100$ $r = 100$ $t_1 = 400$ $t_2 = 200$	$k_1 = 12$ $k_2 = 24$ $p = 90$ $r = 80$ $t_1 = 200$ $t_2 = 200$
Breakout	$k_1 = 12$ $k_2 = 24$ $p = 100$ $r = 100$ $t_1 = 100$ $t_2 = 10$	$k_1 = 12$ $k_2 = 24$ $p = 100$ $r = 100$ $t_1 = 100$ $t_2 = 10$	$k_1 = 12$ $k_2 = 24$ $p = 98$ $r = 80$ $t_1 = 50$ $t_2 = 10$

Table 5.1: Optimal parameters of AD^3 to detect five different attacks.

5.5 Ownership Verification Using Fingerprinting

5.5.1 Model Training

We utilize the same Atari environment, configuration, and DNN architecture as the previous section. We trained six Pong agents for each of the three algorithms DQN, PPO, and A2C for a total of 18 agents. Agents that are trained using the same algorithm use the same hyper-parameters, and the environments are seeded with different random seeds. PPO and A2C agents are trained to follow stochastic policies, and these agents also behave stochastically during test time.

5.5.2 Parameters for ReLF

For **Generate**, we set the size of fingerprint $\mathcal{F}_{\pi_o}$ to $num_f = 5$. $\mathcal{F}_{\pi_o}$ contains five pairs of tuples (r_i, p_i), where r_i is a OSFW(U) perturbation mask and the corresponding SAA p_i.

We generate OSFW(U) adversarial perturbations using randomly selected $k = 60$ states from D_{train} and for $\epsilon = \{0.01, 0.03, 0.05, 0.1\}$. We chose the smallest ϵ for each source policy that can generate fingerprints where SAA is larger than 0.8 when the source policy π_o play against itself. We set $\epsilon = 0.01$ for DQN and PPO agents and $\epsilon = 0.05$ for A2C agent. For **Verify** we set the threshold $\tau_2 = 0.5$ as the threshold value for difference in SAA. To calculate SAA for **Generate** and **Verify**, we use the same parameter values. We use $m = 10$ and $n = 100$ for the number of samples and the sample size and set the threshold value to $\tau_1 = 0.8$. Table 5.2 summarizes all the parameters used for evaluating ReLF.

π_o	ϵ	τ_1	τ_2	m	n
DQN	0.01	0.8	0.5	10	100
A2C	0.05	0.8	0.5	10	100
PPO	0.01	0.8	0.5	10	100

Table 5.2: Parameters of ReLF and its two algorithms **Generate** and **Verify**.

Chapter 6

Evaluation

In this chapter, we evaluate our proposed methods against the various requirements outlined in Chapter 3. In Section 6.1, we evaluate our attacks against the requirements outlined in Section 3.1.3 and argue that prior attack methods do not meet both of these requirements. Next, in Section 6.2 we evaluate our detection method AD^3 against the requirements outlined in Section 3.2.3 and discusses how it can be combined with prior mitigation methods in Section 6.2.2. Finally, Section 6.3 provides an evaluation of our fingerprint method ReLF for ownership verification against the requirements outlined in Section 3.3.4. All of the figures and tables presented in Section 6.1 and Section 6.2 are results that were published as part of our Arxiv paper [68].

6.1 Real-time Adversarial Perturbation Attacks

In this section, we evaluate different adversarial perturbation attacks and discuss how requirements outlined in Section 3.1.3 are fulfilled.

6.1.1 Effectiveness of Adversarial Perturbation Attacks

First, we look at how our attack methods UAP-S, UAP-O, and OSFW(U) satisfy requirement **AR-1** on the adversary *Adv*'s ability to reduce the victim agent v's performance in a task. We compare the effectiveness of our proposed attacks against two baseline attacks FGSM and OSFW, and random noise addition at different ϵ values.

The results are shown in Figure 6.1. Random noise addition has little to no effect on v's average returns except at $\epsilon = 0.01$ for the Breakout PPO agent. FGSM is the most effective attack with 100% performance degradation for all agents in all games at very small ϵ values. UAP-S is the second most effective attack in most settings as it reduces v's returns by 100% at $\epsilon \geq 0.002$ for all Pong agents and at $\epsilon \geq 0.004$ for all Breakout agents. In Freeway, UAP-S reduces v's returns by more than 50% at $\epsilon \geq 0.003$ for all agents.

The attack effectiveness is comparable between OSFW, OSFW(U), and UAP-O. Notably, all attacks can effectively reduce v's returns by 100% at $\epsilon = 0.01$ except against the Freeway PPO agent. In addition, OSFW shows high variability, especially against the Breakout A2C agent and the Freeway PPO agent. While OSFW(U) shows less variability compared to OSFW, both attacks do not satisfy requirement **AR-1** because both attacks cannot fully degrade v's returns in an episode at the largest $\epsilon = 0.01$ against the PPO agent in Freeway. Our proposed attacks UAP-O, UAP-S, and one baseline attack FGSM satisfy requirement **AR-1** because these attacks can reduce v's returns by 100% at $\epsilon = 0.01$.

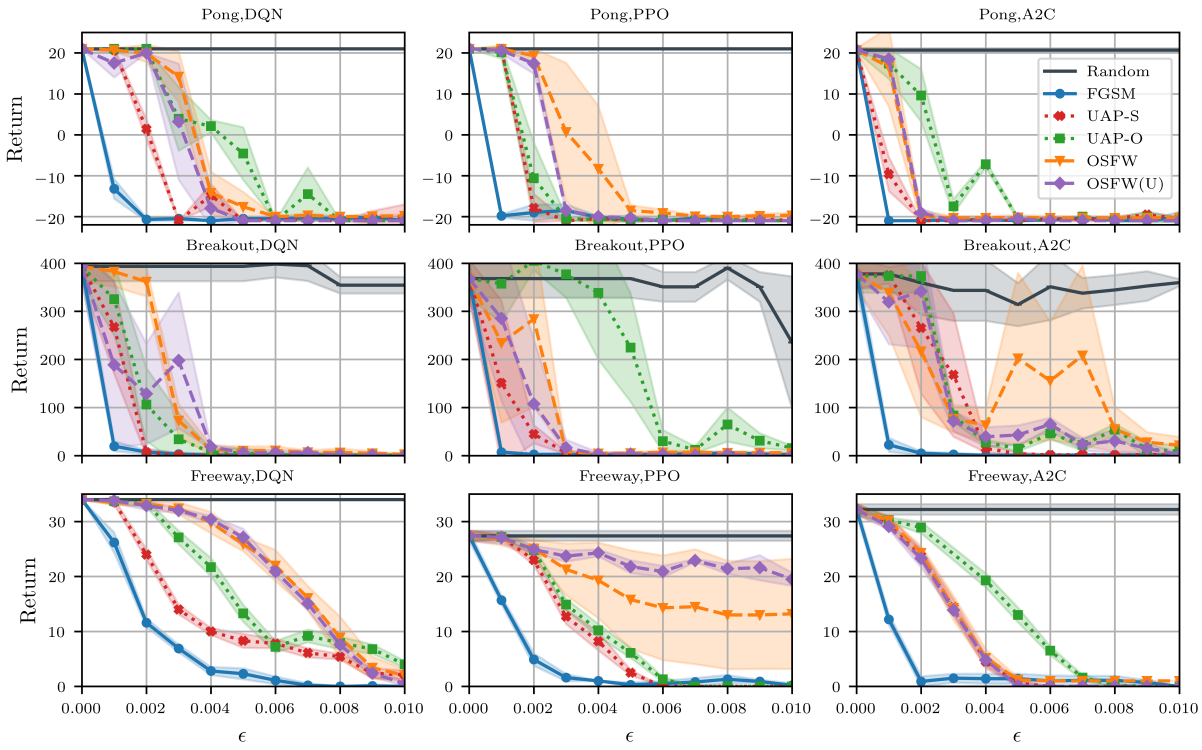


Figure 6.1: Comparison of attacks against three different agents (DQN, PPO, A2C) trained for three different Atari games (Pong, Breakout, Freeway). The graph shows how the returns, averaged over 10 episodes, changes at different ϵ values for six different attacks. The variance of the returns are the shaded region above and below the average values.

6.1.2 Computational Costs of Adversarial Perturbation Attacks

In this section, we discuss whether or not different attacks satisfy requirement **AR-2**, where *Adv*'s online computation cost should be bounded. For *Adv* to mount their attack in real-time, the online computational cost of applying adversarial perturbation r at each time-step is limited by the speed of the environment to move to the next observation plus v's response time.

The maximum upper bound on the online computational time is

$$T_{max} = \frac{1}{\textit{frame rate}} - \textit{agent response time}. \tag{6.1}$$

If the online computational cost is higher than T_{max}, then *Adv* has to stop or delay the environment, which is difficult to do in practice.

Our proposed attacks UAP-S, UAP-O, and OSFW(U) offload computational cost offline by pre-computing r. This offline computation time does not interfere with the real-time requirement of the attack because this offline computation can be carried out before adding r to v's observations of the environment. Therefore, the online computational cost of our proposed attacks only consists of the time to add r to each observation o in the environment. In contrast, FGSM's computational cost is online and it requires computing r based on the current state s of the environment. Similarly, OSFW also computes r online after collecting a number of states from the current episode.

The different computational costs of each attack and the maximum upper bound T_{max} on the online computation time are shown in Table 6.1. The online cost of OSFW is higher than T_{max}, which makes it infeasible to implement in real-time, as mounting this attack requires pausing the environment for more than 5 seconds in the middle of an episode. The online cost of FGSM is lower than T_{max}. However, FGSM requires all observations in a states to arrive before computing r. Therefore, FGSM has to retroactively modify past observations in v's memory. This is beyond the adversary capabilities defined in Section 3.1.2, where *Adv* can only modify observations sequentially and they cannot modify past observations. Therefore, FGSM, while has a acceptable online computational cost, cannot be deployed in real-time.

All of our proposed attacks UAP-S, UAP-O, and OSFW(U), have online computation cost less than T_{max}. This means that all three attacks can be deployed in real-time and satisfy requirement **AR-2**.

Experiment	Attack method	Modify past observations?	Offline cost± std (seconds)	Online cost ± std (seconds)
Pong, DQN, $T_{max} = 0.0163 \pm 10^{-6}$ seconds	FGSM	Yes	-	$13 \times 10^{-4} \pm 10^{-5}$
	OSFW	No	-	5.3 ± 0.1
	UAP-S	No	36.4 ± 21.1	$2.7 \times 10^{-5} \pm 10^{-6}$
	UAP-O	No	138.3 ± 25.1	$2.7 \times 10^{-5} \pm 10^{-6}$
	OSFW(U)	No	5.3 ± 0.1	$2.7 \times 10^{-5} \pm 10^{-6}$
Pong, PPO, $T_{max} = 0.0157 \pm 10^{-5}$ seconds	FGSM	Yes	-	$21 \times 10^{-4} \pm 10^{-5}$
	OSFW	No	-	7.02 ± 0.6
	UAP-S	No	41.9 ± 16.7	$2.7 \times 10^{-5} \pm 10^{-6}$
	UAP-O	No	138.3 ± 25.1	$2.7 \times 10^{-5} \pm 10^{-6}$
	OSFW(U)	No	7.02 ± 0.6	$2.7 \times 10^{-5} \pm 10^{-6}$
Pong, A2C, $T_{max} = 0.0157 \pm 10^{-5}$ seconds	FGSM	Yes	-	$21 \times 10^{-4} \pm 10^{-5}$
	OSFW	No	-	7.2 ± 1.1
	UAP-S	No	11.4 ± 4.3	$2.7 \times 10^{-5} \pm 10^{-6}$
	UAP-O	No	55.5 ± 29.3	$2.7 \times 10^{-5} \pm 10^{-6}$
	OSFW(U)	No	7.2 ± 1.1	$2.7 \times 10^{-5} \pm 10^{-6}$

Table 6.1: Offline and online computation costs of attacks and the maximum upper bound on the time to generate and mount adversarial perturbation attacks during deployment averaged over 10 episodes. Victim agents are DQN, PPO, and A2C trained for Pong. Attacks are deployed with $\epsilon = 0.01$. Attacks that cannot be implemented in real-time are highlighted in red.

6.1.3 Prior Defense in DRL

We evaluate different attacks with respect to requirement **AR-3** by evaluating the effectiveness of the attacks against two defense mechanisms, Visual Foresight and SA-MDP.

As established in Section 6.1.1, except for the case of the Freeways PPO agent, all attacks can reduce v's returns by 100%. To further evaluate the robustness of prior defenses, we evaluate these defense methods at $\epsilon = 0.01$ along with higher ϵ values of 0.02 and 0.05, the results are shown in Table 6.2.

Visual Foresight can recover v's returns for FGSM and UAP-O in both Pong and Freeway for all ϵ. However, it is less effective against OSFW, OSFW(U), and UAP-O. SA-MDP is effective in recovering v's returns at $\epsilon = 0.01$ in both Pong and Freeway. However, SA-MDP's ability to recover v's returns reduces for $\epsilon > 0.01$ and SA-MDP cannot recover v's returns in Pong at $\epsilon = 0.05$. Additionally, SA-MDP lowers v's returns in normal games of Freeway.

Both mitigation techniques cannot fully mitigate the effect of OSFW, UAP-S, UAP-O, and OSFW(U) across all environments at every ϵ. Therefore, we conclude that requirement **AR-3** is partially satisfied for these four attacks. In contrast, FGSM does not satisfy requirement **AR-3** because Visual Foresight can fully recover v's returns in all scenarios.

		Average return ± std in the presence of adversarial perturbation attacks					
ϵ	**Defense**	**No attack**	**FGSM**	**OSFW**	**UAP-S**	**UAP-O**	**OSFW(U)**
0.01	No defense	21.0 ± 0.0	$\mathbf{-21.0 \pm 0.0}$	-20.0 ± 3.0	$\mathbf{-21.0 \pm 0.0}$	-19.8 ± 0.4	$\mathbf{-21.0 \pm 0.0}$
	Visual Foresight [39]	21.0 ± 0.0	21.0 ± 0.0	-19.7 ± 0.5	0.7 ± 1.7	0.4 ± 2.7	$\mathbf{-21.0 \pm 0.0}$
	SA-MDP [81]	21.0 ± 0.0	21.0 ± 0.0	21.0 ± 0.0	21.0 ± 0.0	21.0 ± 0.0	21.0 ± 0.0
0.02	No defense	21.0 ± 0.0	-19.9 ± 1.3	$\mathbf{-21.0 \pm 0.0}$	-20.8 ± 0.6	-20.0 ± 0.0	$\mathbf{-21.0 \pm 0.0}$
	Visual Foresight [39]	21.0 ± 0.0	21 ± 0.0	-19.7 ± 0.6	9.4 ± 0.8	5.3 ± 3.9	-20.5 ± 0.5
	SA-MDP [81]	21.0 ± 0.0	-14.6 ± 8.8	-20.5 ± 0.5	-20.6 ± 0.5	-20.6 ± 0.5	$\mathbf{-21.0 \pm 0.0}$
0.05	No defense	21.0 ± 0.0	-20.5 ± 0.7	$\mathbf{-21.0 \pm 0.0}$	-20.6 ± 0.8	-20.0 ± 0.0	$\mathbf{-21.0 \pm 0.0}$
	Visual Foresight [39]	21.0 ± 0.0	21.0 ± 0.0	-20.0 ± 0.0	7.6 ± 4.7	-14.1 ± 1.1	$\mathbf{-21.0 \pm 0.0}$
	SA-MDP [81]	21.0 ± 0.0	$\mathbf{-21.0 \pm 0.0}$	$\mathbf{-21.0 \pm 0.0}$	-20.6 ± 0.5	-20.6 ± 0.5	$\mathbf{-21.0 \pm 0.0}$

(a) DQN agent playing Pong

		Average return ± std in the presence of adversarial perturbation attacks					
ϵ	**Defense**	**No attack**	**FGSM**	**OSFW**	**UAP-S**	**UAP-O**	**OSFW(U)**
0.01	No defense	34.0 ± 0.0	$\mathbf{0.0 \pm 0.0}$	2.0 ± 1.1	2.1 ± 0.8	4.0 ± 0.6	0.5 ± 0.5
	Visual Foresight [39]	32.0 ± 1.5	32.6 ± 1.7	24.1 ± 1.0	22.9 ± 0.9	25.8 ± 1.1	$\mathbf{20.9 \pm 1.2}$
	SA-MDP [81]	30.0 ± 0.0	30.0 ± 0.0	30.0 ± 0.0	30.0 ± 0.0	30.0 ± 0.0	30.0 ± 0.0
0.02	No defense	34.0 ± 0.0	$\mathbf{0.0 \pm 0.0}$	1.0 ± 0.0	0.1 ± 0.3	0.8 ± 0.6	$\mathbf{0.0 \pm 0.0}$
	Visual Foresight [39]	32.0 ± 1.5	32.6 ± 1.7	$\mathbf{1.1 \pm 0.3}$	24.0 ± 2.0	25.6 ± 1.0	4.4 ± 1.1
	SA-MDP [81]	30.0 ± 0.0	29.8 ± 0.6	29.9 ± 0.3	$\mathbf{29.4 \pm 1.2}$	$\mathbf{29.4 \pm 1.2}$	30.0 ± 0.0
0.05	No defense	34.0 ± 0.0	$\mathbf{0.0 \pm 0.0}$	1.2 ± 0.0	2.2 ± 1.7	2.2 ± 1.4	$\mathbf{0.0 \pm 0.0}$
	Visual Foresight [39]	32.0 ± 1.4	32.6 ± 1.6	1.0 ± 0.0	29.0 ± 1.1	23.9 ± 0.3	$\mathbf{0.0 \pm 0.0}$
	SA-MDP [81]	30.0 ± 0.0	21.1 ± 1.3	$\mathbf{20.9 \pm 0.8}$	21.1 ± 1.7	21.1 ± 1.7	21.1 ± 1.7

(b) DQN agent playing Freeway

Table 6.2: Average returns over 10 episodes with different adversarial perturbation attacks and with victim agents equipped with different types of defenses. In each row, the best attack (lowest return) is in bold font. In each cell, i.e., for a given attack and a given ϵ, the defense that can fully recover the victim agent's returns is shaded green. A cell is shaded blue for the most robust (highest return) defense for that particular attack if it cannot fully recover the victim agent's returns.

6.1.4 Adversarial Perturbation Attacks in the Continuous Control Setting

We show the effectiveness of each attack against PPO agents in Humanoid and Wallker2d environment at different ϵ values in Figure 6.2. FGSM is the most effective attack against the PPO agent in Humanoid environment and is the most effective attack for $\epsilon < 0.12$. UAP-S is able to drastically reduce agent performance for Walker2d and is more effective than FGSM for $\epsilon > 0.12$. Overall, all attacks are effective in reducing v returns. These results demonstrate that all proposed attacks can be generalized to continuous control environments.

Notably, *Adv* optimizes their adversarial perturbation r using the value function V_v. The value function itself can only approximate the value of a state s and lowering the value of a state does not necessarily leads to v choosing less optimal actions. Therefore, *Adv* requires higher ϵ values to reduce v's returns. Table 6.3 shows similar results as Table 6.1. FGSM, UAP-S(O) , and OSFW(U) have online computational time that is less than T_{max} and OSFW is too slow to be mounted in real-time. However, different from Table 6.1, the PPO agents that we downloaded for Walker2d and Humanoid tasks are trained in an environment where each state contains only a single observation. Therefore, FGSM does satisfy requirement **AR-2** in this case because FGSM modifies each observation sequentially. Therefore, the attacks in the continuous control setting still satisfy the same attack requirements as in the discrete environment.

Experiment	**Attack method**	**Offline cost± std (seconds)**	**Online cost ± std (seconds)**
Walker2d, PPO, $T_{max} = 0.0079 \pm 10^{-5}$ seconds	FGSM	-	$31 \times 10^{-5} \pm 10^{-5}$
	OSFW	-	0.02 ± 0.001
	UAP-S(O)	8.75 ± 0.024	$2.9 \times 10^{-5} \pm 10^{-6}$
	OSFW(U)	0.02 ± 0.001	$2.9 \times 10^{-5} \pm 10^{-6}$
Humanoid PPO, $T_{max} = 0.0079 \pm 10^{-6}$ seconds	FGSM	-	$35 \times 10^{-5} \pm 10^{-5}$
	OSFW	-	0.02 ± 0.001
	UAP-S(O)	35.86 ± 0.466	$2.4 \times 10^{-5} \pm 10^{-6}$
	OSFW(U)	0.02 ± 0.001	$2.4 \times 10^{-5} \pm 10^{-6}$

Table 6.3: Offline and online computation cost of attacks and the maximum upper bound for the perturbation generation and mounting the attack during deployment averaged over 10 episodes. Victim agents are PPO agents for Walker2d and Humanoid at $\epsilon = 0.02$. Attacks that cannot be implemented in real-time are highlighted in red.

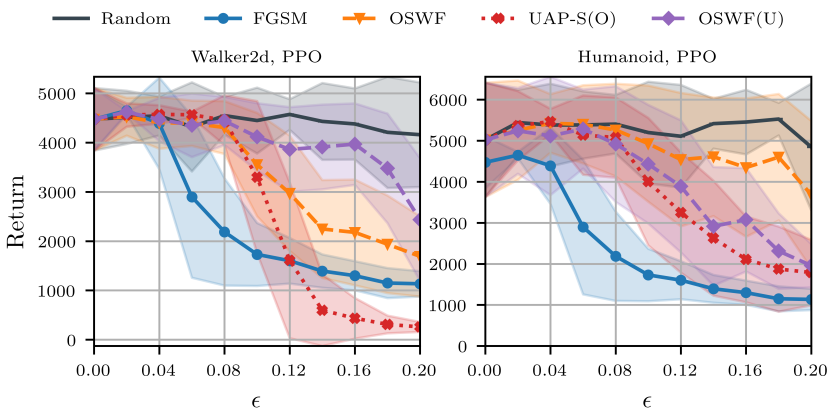


Figure 6.2: Comparison of attacks against PPO agents for Humanoid and Walker-2d tasks. The graph shows how the returns averaged over 50 games at different ϵ values for five different attacks. The variance of the returns are the shaded region above and below the average values.

6.1.5 Summary of Attack Methods

In summary, out of the three proposed attacks, UAP-S and UAP-O are effective in reducing v's returns at $\epsilon = 0.01$ and thus satisfy requirement **AR-1**. Only our three proposed attacks UAP-S, UAP-O, and OSFW(U) can be mounted in real-time (requirement **AR-2**) because the online computation time of the attacks are within the time limit and the attacks does not require modifying v's memory of past observations. None of the attacks can completely evade existing defense mechanisms while some attacks can partially retain its damage to v's returns. FGSM does not satisfy requirement **AR-3** because it can be completely thwarted by Visual Foresight. Other attack, UAP-S, UAP-O, OSFW, and OSFW(U) only partially satisfy **AR-3** because the two prior defense mechanisms cannot fully recover v's returns in all scenarios.

As mentioned in Section 5.2, in our setup for all three Atari games, a single state s is composed of four consecutive observations o of the environment. The baseline attacks FGSM, OSFW, and two of our proposed attacks UAP-S, and OSFW(U) generates four different masks for each of the observations in a single state. On the other hand, UAP-O generates a single perturbation mask for all four observations in a state. The properties of attacks and whether they satisfy the requirements are summarized in Table 6.4.

In Section 6.1.4, we evaluate the effectiveness of all attacks in the continuous control setting. Our experiments show that all attacks require higher ϵ values to be effective.

However, our proposed attacks are still effective and can be mounted in real-time.

Attack Method	Online cost	State dependency	Observation dependency	AR-1	AR-2	AR-3
FGSM [25]	Low	Dependent	Dependent	Yes	No*	No
OSFW [77]	High	Independent	Dependent	No	No	Partial
UAP-S	Low	Independent	Dependent	Yes	Yes	Partial
UAP-O	Low	Independent	Independent	Yes	Yes	Partial
OSFW(U)	Low	Independent	Dependent	No	Yes	Partial

* When deployed in an environment where each state contains more than one observation.

Table 6.4: Summary of five attacks based on the characteristics of the attack. We also summarize which requirements outlined in Section 3.1.3 are met by each attack.

6.2 Detecting Adversarial Perturbation Attacks

In this section, we evaluate our proposed detection technique AD^3 against the detection requirements outlined in Section 3.1.3. We also discuss how AD^3 can be combined with other mitigation methods to improve v's defense in Section 6.2.2.

6.2.1 Evaluating the Effectiveness of AD^3

We evaluate AD^3 with respect to requirement **DR-1** by evaluating the false positive rate and true positive rate of AD^3 in three Atari games, Pong, Freeway, and Breakout against all attacks. Table 6.5 outlines the true positive rate of different attacks at $\epsilon = 0.01$ together with the false positive rate of AD^3 for each configuration. AD^3 can perfectly detect the presence of *Adv* with true positive rate of 1.0 and false positive rate of 0.0 in Pong. However, there are some combinations of environment and agents, where AD^3 is less effective.

As shown in Figure 3.1, OSFW and OSFW(U) are not very effective against the Freeway PPO agent. Additionally, the action change rate, i.e., the percentage of v actions changed by an attack, is low for both UAP-S and UAP-O in Freeway. The action change rate can be as low as $20\% - 30\%$ in some Freeway episodes. Because AD^3 models the distribution of v's conditional action probability distribution (CAPD) during normal episodes, it cannot

detect *Adv* if there are not enough actions changed by the attack in an episode. Attacks with a low action rate result in a less effective attack but are also less detectable by AD^3.

AD^3 has high false positive rates in Breakout compared to other environments. Episodes in Breakout terminate very quickly (as fast as in 112 time-steps) when v is under attack. As a result, AD^3 has to detect the presence of *Adv* early in a Breakout episode. To detect *Adv*, AD^3 cannot skip too many actions at the beginning of an episode. However, AD^3 relies on modeling the CAPD of the current episode, and it takes time to converge. Before CAPD converges, the anomaly score is high for any episode, including episodes with no attack. This leads to a high false positive rate in Breakout. When an episode terminates too quickly in Breakout, AD^3 cannot store enough actions for CAPD to converge and AD^3 did not have time to raise an alarm during the episode.

For episodes where an alarm is raised, the detection mechanism raises the alarm as soon as CAPD of the episode exceeds the threshold th. Therefore, AD^3 satisfies **DR-2**.

In conclusion, AD^3 has varying effectiveness in different environments against different attacks. AD^3 satisfies requirement **DR-1** for all Pong agents. It satisfies requirement **DR-1** for Freeway agents but is less robust when the attacks are not effective. However, requirement **DR-1** is not satisfied for Breakout agents because the false positive rate is high and AD^3 cannot effectively detect some adversarial perturbation attacks.

Game	Agent	FPR	TPR of adversarial perturbation attacks				
			FGSM	OSFW	UAP-S	UAP-O	OSFW(U)
Pong	DQN	0.0	1.0	1.0	1.0	1.0	1.0
	A2C	0.0	1.0	1.0	1.0	1.0	1.0
	PPO	0.0	1.0	1.0	1.0	1.0	1.0
Freeway	DQN	0.0	0.8	1.0	1.0	1.0	0.8
	A2C	0.0	1.0	1.0	1.0	1.0	1.0
	PPO	0.0	1.0	0.4	1.0	1.0	0.6
Breakout	DQN	0.6	1.0	0.6	1.0	1.0	1.0
	A2C	0.0	1.0	0.6	1.0	0.8	1.0
	PPO	0.4	1.0	0.4	1.0	0.6	1.0

Table 6.5: False positive rate and true positive rate of AD^3 against different adversarial perturbation attacks at $\epsilon = 0.01$ over 10 episodes. In each row, attacks with the lowest true positive rate for each victim agent are shaded red. Agents with a none-zero false positive rate are shaded yellow.

6.2.2 Effectiveness of Combining AD^3 with Recovery Methods

In some situations, combining a detection mechanism with a mitigation mechanism can make the defense more effective in preventing *Adv* from achieving requirement **AR-1**. For example, in an environment where v can arrive at a negative outcome as a result of an attack, a detection mechanism can be used to detect the presence of *Adv* early and prevent this negative outcome.

To illustrate this idea, we designed an experiment using a Pong DQN agent because Pong has a clearly defined negative outcome, i.e., when v lose the game. In this setting, an episode ends with a loss if 1) the computer reaches the final score of 21 first, or 2) AD^3 did not raise an alarm before the episode ends.

Table 6.6 presents the *losing rate* of each attack, i.e., the fraction of episodes where v loses the game. As shown in this table, AD^3 can detect the presence of *Adv* for all attacks. In contrast, Visual Foresight cannot recover v's losing rate for OSFW and OSFW(U) at all, and can only recover the losing rate for some episodes of UAP-O. SA-MDP is effective in reducing the losing rate at $\epsilon = 0.01$; however, it fail against all attacks at $\epsilon = 0.02$.

Although detection and recovery are two different aspects of defense, our evaluation shows that utilizing both in combination with tasks with negative outcomes can further thwart *Adv*'s attack effectiveness.

			Losing rate of adversarial attacks				
ϵ	**Method**	**No attack**	**FGSM**	**OSFW**	**UAP-S**	**UAP-O**	**OSFW(U)**
0.01	No defense	0.0	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	Visual Foresight [39]	0.0	0.0	**1.0**	0.0	0.2	**1.0**
	SA-MDP [81]	0.0	0.0	0.0	0.0	0.0	0.0
	AD^3	0.0	0.0	0.0	0.0	0.0	0.0
0.02	No defense	0.0	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	Visual Foresight [39]	0.0	0.0	**1.0**	0.0	0.3	**1.0**
	SA-MDP [81]	0.0	0.9	**1.0**	**1.0**	**1.0**	**1.0**
	AD^3	0.0	0.0	0.0	0.0	0.0	0.0

Table 6.6: Losing rate of DQN agents playing Pong with or without additional defense or detection method for 10 episodes. The losing rate is calculated by counting the number of games where the computer arrives at the maximum score in an episode. If AD^3 raises an alarm before an episode ends, then v does not lose the game. In each row, the best attack with the highest losing rate is in bold font. For a given ϵ value of each attack, the defense with the highest losing rate for that particular attack is shaded red.

6.3 Ownership Verification using Fingerprinting

In this section, we evaluate our fingerprint method ReLF against the requirements outlined in Section 3.3.4. Specifically, we focus on requirement **VR-1** on maximizing true positives and minimizing false negatives.

We have three source policies using three algorithms (DQN, A2C, PPO), and we evaluate ReLF for each source policy π_o against 15 independently trained policies to calculate the false positive rate of the method. Due to time constraints, we are not yet reporting the evaluation of ReLF with respect to the robustness requirement **VR-2**. We will return to it in Section 8.2.1.

ReLF first uses **Generate** to generate a set of fingerprints $\mathcal{F}_{\pi_o}$ from π_o. Next, given a suspect agent *sp* with policy π_s, **Verify** confirms or rejects π_s to be a copy of π_o.

The OSFW(U) masks in $\mathcal{F}_{\pi_o}$ are specifically chosen such that mounting each mask in an episode leads to a high SAA (above 0.8) between π_o and its copies. The SAA of each mask for each π_o can be found in Table 6.7. As the SAA of π_o and itself is high, the algorithm can verify the ownership of all copies of π_o.

Figure 6.3 shows the false positive rate and true negative rate of using different number of OSFW(U) masks in $\mathcal{F}_{\pi_o}$. The false positive rate is 0.0 for $\mathcal{F}_{\pi_o}$ of size $num_f \geq 4$ for all three source policies.

The results suggest that we need more than one OSFW(U) mask in $\mathcal{F}_{\pi_o}$ because OSFW(U) masks occasionally transfer between policies that lead to an independently trained policy to have a high SAA with π_o. As shown in Table 6.7, in the worse case, two out of five OSFW(U) masks would falsely confirm an independently trained policy as a copy og π_o. However, this worse case rarely occurs, as shown in Figure 6.3, the true negative rate of ReLF is higher than 90% for all source policies by using only two OSFW(U) masks in $\mathcal{F}_{\pi_o}$.

Therefore, ReLF satisfies requirement **VR-1** because it can effectively confirm copies of π_o and it can also reject all independently trained policies when we use more than four OSFW(U) masks in $\mathcal{F}_{\pi_o}$.

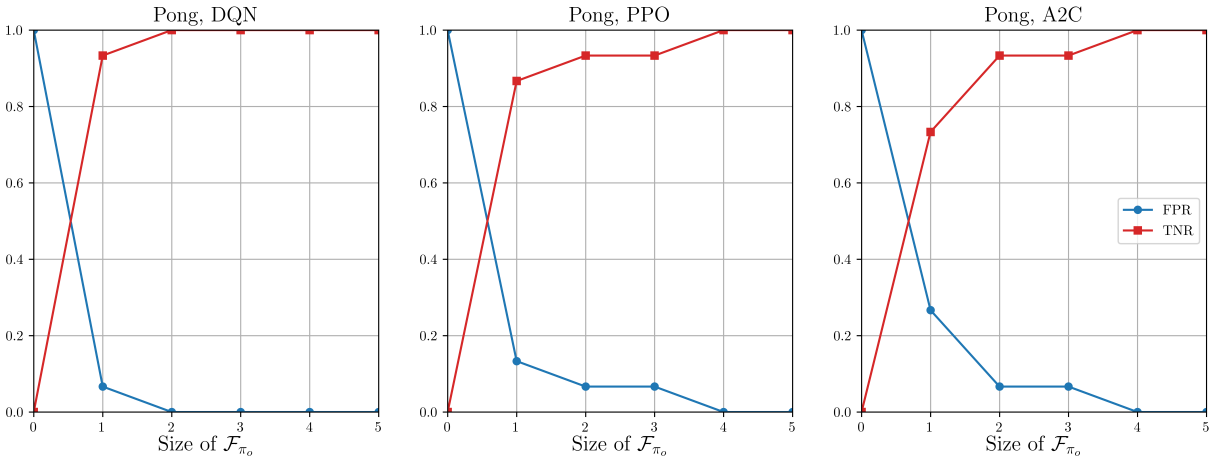


Figure 6.3: The false positive rate and true negative rate of the fingerprint algorithm for each Pong source policies over different numbers of masks in $\mathcal{F}_{\pi_o}$. The masks in $\mathcal{F}_{\pi_o}$ are applied individually to single Pong episodes for each of the 15 individually trained policies.

π_o	**SAA between π_o for episodes with each mask in $\mathcal{F}_{\pi_o}$ applied**	**Min. # masks rejecting π_s**	**Avg. # masks rejecting π_s**
DQN	$\{1.0, 1.0, 1.0, 1.0, 1.0\}$	4	4.93
PPO	$\{0.91, 0.83, 0.91, 0.86, 0.82\}$	3	4.8
A2C	$\{0.81, 0.81, 0.92, 0.89, 0.91\}$	3	4.67

Table 6.7: The SAA between each Pong source policy π_o and its copy where the masks of $\mathcal{F}_{\pi_o}$ are applied individually to an episode, for a total of five episodes. Masks are also applied to the 15 independently trained policies π_s, for a total of five episodes per π_s. For the five episodes testing each π_s, the number of masks rejecting each π_s is recorded. The minimum and the average number of masks of $\mathcal{F}_{\pi_o}$ rejecting π_s are reported.

Chapter 7

Related Work

In this chapter, we discuss related work in adversarial perturbation attacks and ownership verification in deep reinforcement learning (DRL). Section 7.1 discusses adversarial perturbation attacks in different adversary settings. Section 7.2 discusses a model stealing technique and other ownership verification techniques in DRL.

7.1 Other Adversarial Perturbation Attacks in DRL

There are other adversarial perturbation attacks against DRL agents with different configurations and setups. We focus on untargeted white-box adversarial perturbation attacks in this thesis. In this section, we discuss other adversarial perturbation attacks settings. Section 7.1.1 focuses on targeted attacks. Section 7.1.2 discusses attacks where the adversary *Adv* has black-box knowledge of the victim agent v. Finally, Section 7.1.3 presents adversarial perturbation attacks in multi-agent settings.

7.1.1 Targeted Attacks

In this thesis, we focus on untargeted attacks, where *Adv*'s goal is to degrade v performance in a task by misleading v into taking any sub-optimal actions during the task. In contrast, targeted attacks lure v into taking specific actions on specific states and not just any sub-optimal actions. In this case *Adv*'s goal is to either lure v into *Adv*'s desired states or luring v into following an *Adv* defined policy.

Lin et al. [38] propose an enhanced attack where *Adv* lures v into the desired state after a number of steps. *Adv* first trains a future state prediction model based on the work by Oh et al. [51]. Utilizing this prediction model and an action planning method, *Adv* creates a series of adversarial perturbations to lure v into *Adv*'s desired state.

On the other hand, some targeted attack methods [4, 71, 26] focus on utilizing targeted adversarial perturbations to lead v into following an adversarial policy defined by *Adv*. Behzadan and Munir [4] utilize JSMA, and Tretschk et al. [71] utilize Adversarial Transformer Network (ATN) [2] to craft these targeted adversarial perturbations. Hussenot et al. [26] propose CopyCAT. In CopyCAT, multiple adversarial perturbations are pre-computed to manipulate v into taking specific actions.

While targeted attacks can also achieve the goal of lowering v's return in a task, they can also be used to achieve more sophisticated goals such as luring an autonomous vehicle to crash into a specific target on the road. However, some attacks [38, 4] cannot be deployed in real-time because they require pausing the environment, and calculate the adversarial perturbations by sending v with multiple versions of the same state during deployment.

7.1.2 Black-box Attacks

Adv with white-box access to v has knowledge of the weights and the algorithm of v's policy π_v. In contrast, *Adv* with black-box access to v does not have prior knowledge nor access to v's internal states.

Many black-box attacks exploit the transferability of adversarial perturbations between DRL policies where *Adv* first trains a surrogate policy and searches for transferable adversarial perturbations. Huang et al. [25] are the first to study the transferability of FGSM across different DRL algorithms and policies. Their results show that transferability across different DRL algorithms is the least effective in their experiment; however, for most Atari games, adversarial perturbations that transfer across algorithms are still able to decrease v's overall returns. Zhao et al. [84] assume that *Adv* can only observe environment states. They utilize a seq2seq model to approximate π_v and adversarial perturbations are generated from this seq2seq model using FGSM and PGD.

Some black-box attacks also assume that *Adv* has limited knowledge of the environment. In all DRL tasks, the agent and the environment exchange three different types of signals: states, actions, and rewards. In practice, these signals can come from different sources, and *Adv* may only have access to a subset of these sources. Let's consider a real-world example of a DRL agent that controls an autonomous vehicle. The state information is collected through a set of sensors in the vehicle. There is also a remote server running a DRL policy

that takes sensor inputs and outputs action signals back to the vehicle. The reward signals are calculated separately by other sources based on the states. Because these signals are collected through different sources, it is possible that *Adv*'s capability is limited and only has access to a subset of these sources. Inkawhich et al. [27] propose different adversarial perturbation attacks where *Adv* has black-box access to π_v and *Adv*'s knowledge of the environment is a subset of state, action, and reward. Their methods train proxy policies using the same DNN architecture as in Wang et al.'s work [73] on tasks that are similar to v's task and find transferable adversarial perturbations between them.

There are also other black-box attacks where the computation is not based on surrogate policies. Xiao et al. [77] propose different black-box attacks by assuming that *Adv* can perform an unlimited number of queries on the victim network. Their attack utilizes the finite-difference (FD) method to estimate the gradient used to compute adversarial perturbations. Qu et al. [55] estimate the discrepancy of π_v's output action distribution between the original input state versus the perturbed state. They utilize the genetic algorithm to optimize this discrepancy measure to find adversarial perturbations. Notably, both of these black-box attacks cannot be mounted in real-time because they require pausing the environment and querying v multiple times.

The attack methods proposed in this thesis require a powerful adversary with white-box access to v, this may not be realistic in some settings as weights and training algorithms of a DRL agent are often protected and kept secret. In contrast, environment states and action outputs of a DRL agent are more difficult to obscure. Therefore, an adversary with black-box access to v is a more realistic adversary model.

7.1.3 Multi-agent Setting

In multi-agent settings, *Adv* can induce adversarial perturbations as another agent in the environment.

Gleave et al. [19] propose an attack method in this setting by training an adversarial agent that v interacts with within the environment. *Adv* trains the adversarial agent to follow an adversarial policy, where it utilizes the same DRL training algorithm as the victim but with a modified reward function. This reward function rewards the policy that minimizes v's returns. Their experiments show that adversarial policies trained against π_v can effectively defeat v in a task by inducing adversarial observations into the environment using the adversarial agent's body.

Similarly, Wu et al. [76] also propose a method for training adversarial agents in multi-agent environments. They modify the reward function of the adversarial agent to maximize

the deviation of v's output action distributions. They utilize an explainable AI technique to find time-frames during training where v pays the most attention to the opponent agent and utilize these time-frames to maximize divergence in v's output action distribution. Their evaluations show that their attack significantly outperforms Gleave et al.'s attack [19].

Contrasting with our attack setting where *Adv* has to partially control v's observation of the environment, in multi-agent settings, *Adv* only needs to deploy their agent in the environment. Therefore, these attacks in multi-agent settings are more realistic and easier to deploy. However, training DRL agents in multi-agent environments is difficult and time-consuming so we leave expanding our attacks to this setting as future work.

7.2 Ownership Verification in DRL

There are few prior works in DRL on model theft and ownership verification. We first discuss the only work on stealing DRL policies in Section 7.2.1. Then in Section 7.2.2, we discuss two watermark methods.

7.2.1 Stealing DRL Policies

At the writing of this thesis, there is only a single work on stealing DRL policies by Chen et al. [12]. Their technique is split into two parts. The first part is to identify the algorithm used to train π_v, and the second part is on stealing π_v itself. For the first part of their attack, *Adv* trains multiple policies using different algorithms in their own environment. The classifier is then trained to classify a policy to an algorithm based on a sequence of states and action pairs. Once the algorithm of π_v is defined, *Adv* then collects sequences of state and action pairs from v and utilizes Generative Adversarial Imitation Learning (GAIL) [23] to learn π_v. Notably, this technique is resource-intensive where *Adv* has to train many shadow policies for each DRL algorithm to train the algorithm classifier.

7.2.2 Watermarking in DRL

There are only two works on ownership verification in DRL using watermarking techniques.

Behzadan and Hsu [3] utilize states that are disjoint from the training and the deployment environment as the watermarked states. They defined a new reward function and a new state transition function for the watermarked states such that a DRL trained using

these states would act in a specific sequence. During training, the agent alternates between the training environment and watermarked states. During verification, the verifier deploys the suspect agent in an environment that consists of watermarked states and the state transitions are recorded to verify the identity of suspect agent's policy.

Chen et al. [11] argue that it may be impossible in some settings to find watermarked states that are disjoint from the training and deployed environment. Instead, they utilize states that naturally occur in the training and deployment environment as the watermarked states. First, the verifier identifies a set of non-critical states. These states are such that an agent can still achieve optimal returns in their task even if non-optimal actions are selected in these states. They take these non-critical states and map them to specific actions that are used to identify a suspect agent's policy during verification.

For both works, watermarking a DRL policy requires modifications to the training of the source policy, whereas fingerprint methods can be applied to any existing DRL policies directly. Additionally, the verifier needs to inject watermarked states into the environment to verify a suspect agent. This requires pausing and modifying environment states to watermarked states. This is difficult in real-time because it is infeasible to change the entire environment state every time-steps. While Chen et al.'s [11] method could be used by deploying the suspect agent in a normal environment and only recording the action when watermarking states naturally occur in the environment, it is not guaranteed that the watermarked states will occur. In contrast, our proposed fingerprint method injects adversarial perturbations to the observations of the environment instead of modifying the environment states completely and can be applied to existing DRL agents.

Chapter 8

Discussion

In this chapter, we discuss future work and improvements for our proposed methods. In Section 8.1 we discuss future work and improvements for our proposed attacks and detection technique. In Section 8.2 we discuss the robustness of our proposed fingerprinting method and future work.

8.1 Real-time Adversarial Perturbation Attack and Defense in DRL

In Section 8.1.1, we discuss future work and improvements for our proposed adversarial perturbation attacks. In Section 8.1.2, we discuss the limitations in using l_p norms for adversarial perturbation attacks. In Section 8.1.3, we discuss the robustness of our detection technique AD^3 and possible improvements to AD^3. Finally, in Section 8.1.4, we discuss how combining different defense methods can better improve the security of deep reinforcement learning (DRL) agents.

8.1.1 Future Work for Adversarial Perturbation Attacks

In Section 6.1.1, we demonstrated that our proposed attacks (UAP-S, UAP-O, and OSFW(U)) are effective in discrete Atari game environments at small ϵ bounds. while our attacks are also effective in continuous control settings, they require significantly higher ϵ bounds for these attacks to be effective. We modified our proposed attacks for continuous control to

use the victim agent v's value function V_v to find a perturbation r that lowers the value of a state s. As r is optimized to lower the value of s, it does not change the output action distribution of v's policy π_v directly. One way to improve these attack methods is to optimize for the divergence in π_v's output action distribution, such that v is more likely to select sub-optimal actions.

The two testing environments that we use are relatively simple compared to environments such as autonomous driving [52, 61] and robotic manipulation [36]. Further complications may arise due to the difference in frame rate and the increase in the dimensions of the input and output to the policy. However, because UAP attacks only require adding a fixed noise mask to the input stream, UAP attacks are fast and can be applied to environments with frame rate that is higher than our testing environment. We leave the evaluation of the generalizability of our attacks to other complex tasks as future work.

We can further expand our attack methods to the black-box setting, i.e., the adversary *Adv* does not have access to weights and the training algorithm of π_v, by first mounting a model extraction attack and finding transferable adversarial perturbations using our techniques.

8.1.2 Limitations of using l_p Norms in Adversarial Perturbation Attacks

It is a standard practice for adversarial perturbation attacks in the image domain to use l_p norms and a threshold value ϵ to constrain the size of the perturbation. In practice, we assume that an adversarial perturbation is undetectable by the human eyes if the size of the perturbation is smaller than ϵ in a l_p norm. However, it is difficult to measure human perceptual similarity, and similarity in the l_p norms is not a necessary and sufficient condition for perceptual similarity.

In particular, prior work [17, 30, 78] shows that two inputs that are very different as measured by a l_p norm does not imply perceptual dissimilarity. For example, an adversarial example constructed through spatial transformations [17], e.g., rotation and translation, and the original input are measured by l_p norms to be very different from each other. However, the two inputs are still perceptually indistinguishable to humans. Furthermore, Sharif et al. [62] show through an online study that adversarial examples constructed though commonly used l_p norm thresholds can still be perceptually dissimilar to the original inputs.

Additionally, l_p norms have limited scope and are not applicable to many of the problem outside of the image processing domain. As an example, we take a look at malware

detection. In malware detection, the inputs are features of discrete values, e.g., the binary of a file. To fool a malware classifier using adversarial examples, it is more important for the adversary to preserve the functionalities of a malware file than to preserve the l_p similarity between the binary of the adversarial example and the binary of the original file. Therefore, to create effective adversarial examples in this domain, an adversary has to define a different similarity measure to achieve their goal successfully.

8.1.3 Improvements for AD^3

As discussed in Section 6.2.1, AD^3 can detect adversarial perturbation attacks that are applied uniformly throughout the episode by AD^3 requires collecting several consecutive anomalous actions before raising an alarm.

A possible evasion technique against AD^3 is to apply adversarial perturbations in short alternating intervals. However, this would create between-observations anomalies, where some observations are clean and others have adversarial perturbations. Visual Foresight is designed to detect and mitigate these between-observation anomalies because the frame prediction module can utilize the clean observations to predict the correct next observation in the environment. As a result, attacks that are not applied uniformly throughout the episode can be mitigated by Visual Foresight. Therefore, combining Visual Foresight and AD^3 can detect short intervals of adversarial perturbations, which AD^3 cannot do alone.

At present, AD^3 is only applicable to environments with discrete action spaces. As shown in Section 6.1.4, DRL agents are also vulnerable to adversarial perturbation attacks in continuous control settings. As future work, we will expand AD^3 to continuous control environments.

8.1.4 Capabilities of Defense Mechanisms

When a defense mechanism is not sufficient on its own, it may be useful to deploy other complementary defense mechanisms at the same time to maximize defense coverage.

As discussed in Section 6.1.3, both prior defense methods, Visual Foresight and SA-MDP, cannot successfully recover v's returns against all attacks across all environments. It may be possible to further improve the recovery of v's returns by deploying both defense mechanisms at the same time. However, as presented in Table 6.2, in the case of a Pong agent against attacks bounded by $\epsilon = 0.05$, only FGSM can be mitigated completely by Visual Foresight and SA-MDP and v's returns still drop significantly against all other

attacks. Therefore, it is still unclear whether adversarial perturbation attacks can be completely mitigated by combining these two methods.

Additionally, in Section 6.2.2 we consider a scenario where *Adv* causes v to arrive at a negative outcome in the environment, e.g., losing a game in Pong. In this case, combining mitigation with detection mechanisms can be useful. As shown in Table 6.6, AD^3 can completely prevent v from losing in Pong games.

In conclusion, while combining different mitigation and detection methods can provide better protection for DRL agents, there are still attacks that cannot be thwarted completely by existing defense mechanisms. Similar to deep neural network (DNN) clssifiers, adversarial perturbations are features of the neural network and are difficult to mitigate without sacrificing accuracy or returns.

8.2 Ownership Verification using Fingerprinting

In Section 8.2.1, we first discuss how *Adv* can evade our fingerprint method ReLF. In Section 8.2.2, we discuss how fingerprint verification can be deployed in multi-agent settings.

8.2.1 Robustness of Fingerprinting

Due to time constraints, we did not report robustness in the evaluation of ReLF (Section 6.3). Our robustness experiments are currently in progress. Instead, in this thesis, we offer discussions on how *Adv* can evade verification.

We evaluate ReLF against suspect agents with stolen policies that are exact copies of the source policy. As mentioned in Section 7.2.1, there is only one work [12] on stealing DRL policies. Unfortunately we cannot find public repository for their work. Stolen policies obtained from model extraction attacks are not exact copies of the source policy. As a result, the effectiveness of adversarial perturbation attacks could be reduced. Because ReLF relies adversarial perturbations' effectiveness on the source policy, ReLF's ability to identify stolen policies could be reduced. We leave implementation and evaluation of ReLF against model extraction attack as future work.

One way for *Adv* to evade verification is to deploy their stolen agent such that it selects random actions for some states. These states can be randomly chosen or they can be non-critical states as described in Section 4.1.2 to limit the effects on returns of the stolen agent. Notably, selecting random actions instead of optimal actions would negatively impact the

stolen agent's returns in a task. As a result of selecting random actions in some states, SAA between the stolen policy and the source policy diverges because the number of actions agreed between the two policies decreases. *Adv* needs to balance this strategy with the loss of utility of the stolen policies because the more random actions taken, the more it negatively impacts the agent's returns in a task.

Additionally, *Adv* can transform their stolen policies to remove fingerprints. Fine-tuning/retraining is a technique that takes the parameters of a source model as a starting point and continues to train the model on a set of data. In DRL, *Adv* can fine-tune a stolen policy in the same training environment as the source policy but with a different seed. Another approach is to do adversarial retraining on the stolen policy. However, adversarial retraining can lead to unstable training and reduced performance.

Finally, another way to evade verification is to deploy the two defense methods discussed in Section 4.1.4. However, as shown in Table 6.2, both Visual Foresight and SA-MDP have limited effectiveness against OSFW(U) attacks. Visual Foresight cannot recover v's returns in Pong at $\epsilon \geq 0.01$ and SA-MDP only barely recovers v's returns at $\epsilon \geq 0.2$. These results suggest that both defense mechanisms cannot mitigate the effect of OSFW(U) attack on a stolen policy and that the stolen policy would still be misled into selecting the same sub-optimal action as the source policy. Therefore, it is unclear as to whether or not *Adv* can evade verification using these two defense methods.

8.2.2 Fingerprinting in the Multi-Agent Setting

Both ReLF and prior work on watermark methods require the verifier to have control over the environment or the suspect agent's observations of the environment, whether it is to control the states of the environment or to inject adversarial perturbations to the suspect agent's observations of the environment.

In multi-agent environments, a deployed DRL agent necessarily interacts with other agents in the environment. In this setting, the verifier can deploy an agent in the environment to interact with the suspect agent to verify fingerprints. In contrast to the previous setting, the verifier does not need to control the environment state or inject noise into the suspect agent's observations; the fingerprints are triggered as a natural part of the suspect agent's observation of the environment. Therefore, multi-agent environments are more realistic settings for ownership verification using fingerprints.

As mentioned in Section 7.1.3, Gleave et al. [19] propose a technique for training adversarial policy against v. Through their experiments, they found that an adversarial agent with this adversarial policy often does not find better strategies in a task; instead, the

adversarial agent manipulates its body to create adversarial perturbations to v's observations of the environment. How v's actions change as a result of the adversarial agent in the environment can be used to fingerprint v's policy because v would behave in a novel way in this scenario. We leave the design of this fingerprinting method for future work.

Chapter 9

Conclusion

In this thesis, we focus on two security concerns in deep reinforcement learning (DRL). The first is on adversarial perturbation attacks against DRL agents. As DRL agents show many successes in safety-critical tasks such as robotic control and autonomous driving, it is crucial to study these vulnerabilities before deploying these agents in practice. The second topic is on ownership demonstration for DRL policies. Model theft and unauthorized distribution of DRL policies constitute a serious threat to policy owners because they can negatively impact policy owners' revenue. We show the seriousness of these two concerns, discuss the limitations of current techniques, and propose our own solutions.

First, we show that prior work on untargeted attacks does not satisfy the requirements for real-time perturbation attacks against DRL policies because they are either too slow to be real-time or they require modifying the victim policy's internal memory of past observations. We propose three different universal perturbation attacks, UAP-S, UAP-O, and OSFW(U) that can be deployed in real-time and in a white-box setting. Our results show that our attacks are effective against three different DRL policies (DQN, A2C, and PPO). We also evaluate our attacks against two prior defense methods, Visual Foresight and SA-MDP. These defense methods can recover victim performance in different environments with varying results. While in some environments these defense methods are effective, for other environments such as a DQN agent playing Pong, three of the attacks, OSFW, UAP-S, and UAP-O can fully reduce victim's returns by 100% at higher ϵ values. We further expand our attacks to the continuous control settings utilizing the victim's value function. We found that our attacks are still effective in continuous control tasks; however, the attacks require higher ϵ bounds. For future work, we will improve the effectiveness of our attack methods to generalize to different continuous control settings and make the attacks more realistic and easier to deploy for the adversary.

We also propose a new detection technique AD^3 to detect the presence of an attack in discrete settings. We evaluate AD^3 in three environments (Pong, Freeway, and Breakout). AD^3 can perfectly detect all attacks in Pong but is less robust in Freeway and is ineffective in Breakout. We argue that in environments that have a negative outcome, AD^3 can be used to detect the attack during an episode and prevent the negative outcome. For future work, we will generalize this defense technique to different continuous control settings.

To address the second security concern in DRL, we propose a fingerprint method ReLF for ownership verification. We argue that prior watermark methods in DRL require a strong assumption that the verifier has complete control over the environment used to verify the suspect policy. ReLF only requires the verifier to inject adversarial perturbations to the suspect policy's observations of the environment. Additionally, our fingerprint method does not require modifying the source policy and can be applied to existing DRL policies.

Our experimental results show that ReLF can reliably identify a stolen policy with a low false positive rate and a high true positive rate by using only a small number of fingerprints. We leave the evaluation of the robustness of the fingerprint method as future work. The adversary can thwart ownership verification attempts on their policies by deploying defense methods against adversarial perturbation attacks or by adding random actions to their deployed policies. For future work, we will expand ReLF into the multi-agent settings where the verifier controls one of the agents in the environment.

In conclusion, we demonstrate that adversarial perturbation attacks can be mounted in real-time and can significantly impact a DRL agent's performance in a task. While existing defenses can mitigate some effects of the attacks, it is still insufficient in recovering the original performance of the agent. Further studies are also needed in more complex control settings. We also show that ReLF is a promising method for ownership verification of DRL policies and is easier to deploy compared to watermark methods. Ownership verification techniques are useful for DRL policy owners to deter and prosecute the unauthorized distribution of their policies.

References

[1] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1615–1631, 2018.

[2] Shumeet Baluja and Ian Fischer. Adversarial transformation networks: Learning to generate adversarial examples. *arXiv preprint arXiv:1703.09387*, 2017.

[3] Vahid Behzadan and William Hsu. Sequential triggers for watermarking of deep reinforcement learning policies. *arXiv preprint arXiv:1906.01126*, 2019.

[4] Vahid Behzadan and Arslan Munir. Vulnerability of deep reinforcement learning to policy induction attacks. In *International Conference on Machine Learning and Data Mining in Pattern Recognition*, pages 262–275. Springer, 2017.

[5] Vahid Behzadan and Arslan Munir. Whatever does not kill deep reinforcement learning, makes it stronger. *arXiv preprint arXiv:1712.09344*, 2017.

[6] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[7] Franziska Boenisch. A survey on model watermarking neural networks. *arXiv preprint arXiv:2009.12153*, 2020.

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[9] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14, 2017.

[10] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 39–57. IEEE, 2017.

[11] Kangjie Chen, Shangwei Guo, Tianwei Zhang, Shuxin Li, and Yang Liu. Temporal watermarks for deep reinforcement learning models. In *Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems*, pages 314–322, 2021.

[12] Kangjie Chen, Shangwei Guo, Tianwei Zhang, Xiaofei Xie, and Yang Liu. Stealing deep reinforcement learning models for fun and profit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 307–319, 2021.

[13] Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. Refit: a unified watermark removal framework for deep learning systems with limited data. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 321–335, 2021.

[14] Kenneth T Co, Luis Muñoz-González, Sixte de Maupeou, and Emil C Lupu. Procedural noise adversarial examples for black-box attacks on deep convolutional networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 275–289, 2019.

[15] Xiaohui Dai, Chi-Kwong Li, and Ahmad B Rad. An approach to tune fuzzy controllers based on reinforcement learning for autonomous vehicle control. *IEEE Transactions on Intelligent Transportation Systems*, 6(3):285–293, 2005.

[16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338. PMLR, 2016.

[17] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. A rotation and a translation suffice: Fooling cnns with simple transformations. In *7th International Conference on Learning Representations*. OpenReview.net, 2019.

[18] Lixin Fan, Kam Woh Ng, and Chee Seng Chan. Rethinking deep neural network ownership verification: Embedding passports to defeat ambiguity attacks. *Advances in Neural Information Processing Systems*, 32, 2019.

[19] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. In *International Conference on Learning Representations*, 2019.

[20] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[21] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[22] Jamie Hayes and George Danezis. Learning universal adversarial perturbations with generative models. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 43–49. IEEE, 2018.

[23] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in Neural Information Processing Systems*, 29:4565–4573, 2016.

[24] Mengdi Huai, Jianhui Sun, Renqin Cai, Liuyi Yao, and Aidong Zhang. Malicious attacks against deep reinforcement learning interpretations. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 472–482, 2020.

[25] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[26] Léonard Hussenot, Matthieu Geist, and Olivier Pietquin. Copycat: Taking control of neural policies with constant attacks. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, 2020.

[27] Matthew Inkawhich, Yiran Chen, and Hai Li. Snooping attacks on deep reinforcement learning. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*, pages 557–565, 2020.

[28] Hengrui Jia, Christopher A Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. Entangled watermarks as a defense against model extraction. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1937–1954, 2021.

[29] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[30] Can Kanbak, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. Geometric robustness of deep networks: analysis and improvement. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4441–4449, 2018.

[31] Jernej Kos and Dawn Song. Delving into adversarial attacks on deep policies. In *5th International Conference on Learning Representations*. OpenReview.net, 2017.

[32] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Anuals of Mathematical Statistics*, 22(1):79–86, 1951.

[33] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *Artificial Intelligence Safety and Security*, pages 99–112. Chapman and Hall/CRC, 2018.

[34] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10(1), 2009.

[35] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial frontier stitching for remote neural network watermarking. *Neural Computing and Applications*, 32(13):9233–9244, 2020.

[36] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[37] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of dnn. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 126–137, 2019.

[38] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. Tactics of adversarial attack on deep reinforcement learning agents. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3756–3762. ijcai.org, 2017.

[39] Yen-Chen Lin, Ming-Yu Liu, Min Sun, and Jia-Bin Huang. Detecting adversarial attacks on neural network policies with visual foresight. *arXiv preprint arXiv:1710.00814*, 2017.

[40] Nils Lukas, Yuxuan Zhang, and Florian Kerschbaum. Deep neural network fingerprinting by conferrable adversarial examples. In *International Conference on Learning Representations*, 2020.

[41] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[42] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147, 2017.

[43] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[46] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1765–1773, 2017.

[47] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[48] Konda Reddy Mopuri, Utsav Garg, and R Venkatesh Babu. Fast feature fool: A data independent approach to universal adversarial perturbations. *arXiv preprint arXiv:1707.05572*, 2017.

[49] Konda Reddy Mopuri, Utkarsh Ojha, Utsav Garg, and R Venkatesh Babu. Nag: Network for adversary generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 742–751, 2018.

[50] Ryota Namba and Jun Sakuma. Robust watermarking of neural network with exponential weighting. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 228–240, 2019.

[51] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

[52] Eshed Ohn-Bar and Mohan Manubhai Trivedi. Looking at humans in the age of self-driving and highly automated vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):90–104, 2016.

[53] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[54] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks. *arXiv preprint arXiv:1712.03632*, 2017.

[55] Xinghua Qu, Zhu Sun, Yew-Soon Ong, Abhishek Gupta, and Pengfei Wei. Minimalistic attacks: How little it takes to fool deep reinforcement learning policies. *IEEE Transactions on Cognitive and Developmental Systems*, 13(4):806–817, 2020.

[56] Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[57] Lalit Kumar Saini and Vishal Shrivastava. A survey of digital watermarking techniques and its applications. *arXiv preprint arXiv:1407.4735*, 2014.

[58] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[59] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[60] Masoumeh Shafieinejad, Nils Lukas, Jiaqi Wang, Xinda Li, and Florian Kerschbaum. On the robustness of backdoor-based watermarking in deep neural networks. In *Proceedings of the 2021 ACM Workshop on Information Hiding and Multimedia Security*, pages 177–188, 2021.

[61] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[62] Mahmood Sharif, Lujo Bauer, and Michael K Reiter. On the suitability of lp-norms for creating and preventing adversarial examples. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1605–1613, 2018.

[63] Ruimin Shen, Yan Zheng, Jianye Hao, Zhaopeng Meng, Yingfeng Chen, Changjie Fan, and Yang Liu. Generating behavior-diverse game ais with evolutionary multi-objective deep reinforcement learning. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, pages 3371–3377, 2020.

[64] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–601, 2017.

[65] Jianwen Sun, Tianwei Zhang, Xiaofei Xie, Lei Ma, Yan Zheng, Kangjie Chen, and Yang Liu. Stealthy and efficient adversarial attacks against deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5883–5891, 2020.

[66] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[67] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[68] Buse GA Tekgul, Shelly Wang, Samuel Marchal, and N Asokan. Real-time attacks against deep reinforcement learning policies. *arXiv preprint arXiv:2106.08746*, 2021.

[69] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[70] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1633–1645. Curran Associates, Inc., 2020.

[71] Edgar Tretschk, Seong Joon Oh, and Mario Fritz. Sequential attacks on agents for long-term adversarial goals. *arXiv preprint arXiv:1805.12487*, 2018.

[72] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 269–277, 2017.

[73] Elias Wang, Atli Kosson, and Tong Mu. Deep action conditional neural network for frame prediction in atari games. Technical report, Stanford University, 2017.

[74] Jiangfeng Wang, Hanzhou Wu, Xinpeng Zhang, and Yuwei Yao. Watermarking in deep neural networks via error back-propagation. *Electronic Imaging, Media Watermarking, Security, and Forensics*, 2020(4):22–1, 2020.

[75] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[76] Xian Wu, Wenbo Guo, Hua Wei, and Xinyu Xing. Adversarial policy training against deep reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1883–1900, 2021.

[77] Chaowei Xiao, Xinlei Pan, Warren He, Jian Peng, Mingjie Sun, Jinfeng Yi, Bo Li, and Dawn Song. Characterizing attacks on deep reinforcement learning. *arXiv preprint arXiv:1907.09470*, 2019.

[78] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. *arXiv preprint arXiv:1801.02612*, 2018.

[79] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2018.

[80] Valentina Zantedeschi, Maria-Irina Nicolae, and Ambrish Rawat. Efficient defenses against adversarial attacks. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 39–49, 2017.

[81] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Mingyan Liu, Duane Boning, and Cho-Jui Hsieh. Robust deep reinforcement learning against adversarial perturbations on state observations. *Advances in Neural Information Processing Systems*, 33:21024–21037, 2020.

[82] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 159–172, 2018.

[83] Jingjing Zhao, Qingyue Hu, Gaoyang Liu, Xiaoqiang Ma, Fei Chen, and Mohammad Mehedi Hassan. Afa: Adversarial fingerprinting authentication for deep neural networks. *Computer Communications*, 150:488–497, 2020.

[84] Yiren Zhao, Ilia Shumailov, Han Cui, Xitong Gao, Robert Mullins, and Ross Anderson. Blackbox attacks on reinforcement learning agents using approximated temporal information. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 16–24. IEEE, 2020.

[85] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784. IEEE, 2019.

Glossary

A2C Advantage Actor-critic. A DRL algorithm by Mnih et al. [43] x, xii, xv, xvi, 3, 7, 22, 30, 31, 38–42, 44, 45, 47, 52, 54, 55, 67

AD^3 Action Distribution Divergence Detector. This is our proposed detection method for detection adversarial perturbation attacks. x, xii, xiii, 3, 27–29, 37, 40, 41, 43, 51–53, 61, 63, 64, 68

DQN Deep Q Network. A DRL algorithm by Mnih et al. [44]. x, xii, xiii, xv, xvi, 3, 7, 22, 28, 29, 38–42, 45, 47, 48, 52–55, 67

FGSM Fast Gradient Sign Method. An untargeted adversarial perturbation generation method by Goodfellow et al. [20]. x, 3, 10, 11, 26, 29, 38, 43, 44, 46–53, 57, 63

OSFW One of Xiao et al. [77]'s propose attack. They named it obs-fgsm-wb in their work. 3, 11, 22, 25, 26, 38, 39, 43, 44, 46–53, 67

OSFW(U) Our modification of Xiao et al.'s [77] attack. 3, 21, 22, 25, 26, 31, 32, 38, 41–44, 46–54, 61, 65, 67

PPO Proximal Policy Optimization. A DRL algorithm by Schulman et al. [58] x, xii, xv, xvi, 3, 7, 22, 30, 31, 38–42, 44, 45, 47, 49, 50, 52, 54, 55, 67

ReLF Reinforcement Learning Fingerprints. This is our proposed fingerprint method. xii, 3, 30–32, 34, 36, 37, 42, 43, 54, 64, 65, 68

SA-MDP A adversarial retraining method for DRL policies by Zhang et al. [81]. 27, 39, 47, 48, 53, 63, 65, 67

SAA Sample Action Agreement. A Measure of the sample action agreement between two action sequences of the same length. xiii, 30–32, 35, 41, 42, 54, 55, 65

UAP-O Observation-agnostic universal adversarial perturbation. One of our propose attack. 3, 21–23, 25, 26, 31, 38, 39, 43, 44, 46–48, 50–53, 61, 67

UAP-S State-agnostic universal adversarial perturbation. One of our propose attack. 3, 21–23, 25, 26, 31, 38, 39, 43, 44, 46–53, 61, 67

Visual Foresight A mitigation and detection technique by Lin et al. [39]. 27, 39, 47, 48, 50, 53, 63, 65, 67