AIRS: Explanation for Deep Reinforcement Learning based Security Applications

Jiahao Yu
Northwestern University

Wenbo Guo
Purdue University

Qi Qin
ShanghaiTech University[*]

Gang Wang
University of Illinois at Urbana-Champaign

Ting Wang
Pennsylvania State University

Xinyu Xing
Northwestern University

Abstract

Recently, we have witnessed the success of deep reinforcement learning (DRL) in many security applications, ranging from malware mutation to selfish blockchain mining. Like all other machine learning methods, the lack of explainability has been limiting its broad adoption as users have difficulty establishing trust in DRL models' decisions. Over the past years, different methods have been proposed to explain DRL models but unfortunately, they are often not suitable for security applications, in which explanation fidelity, efficiency, and the capability of model debugging are largely lacking.

In this work, we propose AIRS, a general framework to explain deep reinforcement learning-based security applications. Unlike previous works that pinpoint important features to the agent's current action, our explanation is at the step level. It models the relationship between the final reward and the key steps that a DRL agent takes, and thus outputs the steps that are most critical towards the final reward the agent has gathered. Using four representative security-critical applications, we evaluate AIRS from the perspectives of explainability, fidelity, stability, and efficiency. We show that AIRS could outperform alternative explainable DRL methods. We also showcase AIRS's utility, demonstrating that our explanation could facilitate the DRL model's failure offset, help users establish trust in a model decision, and even assist the identification of inappropriate reward designs.

1 Introduction

Deep reinforcement learning (DRL) has shown great success in a range of applications such as playing GO [61, 62] and complex video games [48, 72]. The core idea is to train a deep neural network agent that makes a sequence of decisions to achieve its goals based on its observation of the environment; meanwhile, the decisions/actions from the agent will also dynamically affect the environment. Recently, researchers start to apply DRL in security- and safety-critical applications by formulating them as a sequential decision-making problem [1, 5, 16, 27, 29, 44, 45, 80]. Examples of such applications include malware mutation [9, 44, 78, 79, 82], network lateral movement attack and defense [45], blockchain mining [13], and autonomous driving [37].

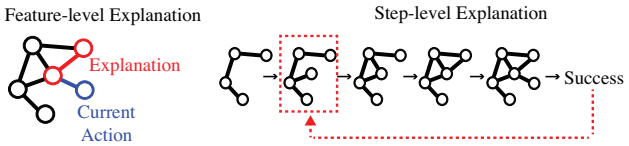


Figure 1: **Example**—Two types of explanations for malware mutation using DRL on a call graph. Feature-level explanation (left) identifies important features for the agent's decision at a given moment. Step-level explanation (right) identifies the most critical step(s) that lead to final evasion results.

While the performance of DRL is promising, it is often difficult for humans to understand the agent's decisions computed by a deep policy network. The absence of explanations of DRL models creates a key barrier to establishing trust in the DRL agents' decisions. Furthermore, without an explanation mechanism, it is difficult to incorporate the black-box DRL model with expert knowledge to enable policy debugging or security analysis for security- and safety-critical applications.

Recent works have proposed explanation methods for DRL agents, but they suffer from three main limitations when applied to security applications. *First*, most existing methods focus on the feature-level explanation [7, 11, 39, 40, 43, 71] to explain an agent's action *at a given step*. However, such explanations provide little insight into why the agent would eventually succeed/fail in the overall task. For example, Figure 1 illustrates an example of DRL-based malware mutation [83]. The DRL agent takes the call graph of a malware sample as the input and takes a series of actions to mutate the call graph step by step to generate a sample that can evade detection (see more details in Section 5). Feature-level explanation methods (left) aim to identify a small set of features (i.e., a sub-graph) in the current call graph to explain the DRL agent's manipulation decision at this step. While this explanation helps to interpret the agent's action of this specific

[*]Work done while at Northwestern University.

step, it is not helpful to understand why the agent would succeed or fail the evasion in the end. *Second*, other explanation methods utilize value functions of the target agent to assess state importance [4, 31]. As we will show in Section 6, these methods cannot provide high-fidelity explanations for DRL-based security applications. Besides, they are not applicable to cases where value functions are not available. *Third*, one recent method focus on the coarse-grained *global* explanations [22]. The explanation results are too generic to debug individual runs of the DRL agent. For example, for malware mutation, the globally-important mutations may not inform why a specific malware sample's mutation process succeeded (or failed). *Finally*, explanation methods for security applications have higher expectations not only for the explanation fidelity but also for explanation stability and efficiency [75], which are not the main focus of existing works.

In this paper, we propose AIRS: "**A**pproximation-based **I**nterpreter for Deep **R**einforcement Learning in **S**ecurity Applications", which is a general framework to explain the policy of DRL model. AIRS is a *local* explanation method for explaining individual episodes (samples). Instead of focusing on feature-level explanation, our goal is to provide the *step-level explanation* to identify the critical steps that contribute the most to the final reward in a given run. In the example shown in Figure 1 (right), the explanation method pinpoints the most critical steps to the success/failure of the malware mutation. Once the critical steps are identified, analysts can further inspect the actions taken in these steps to debug the DRL agent's policy network. Furthermore, to produce high-fidelity, stable and efficient explanations, we develop a novel architecture to approximate the final reward for the DRL agent. For this architecture, we use a recurrent neural network (RNN) to capture the temporal relationship between states while using fully-connected deep neural networks to approximate the reward function. The approximation model makes it possible to obtain the association between each step and the final reward.

We evaluate the performance of AIRS using four different security- and safety-critical applications driven by DRL, namely, autonomous driving [37], malware mutation [83], selfish blockchain mining [13], and network lateral movement optimization [45]. We show that AIRS outperforms existing (baseline) DRL explanation methods in terms of explanation fidelity, stability, and efficiency. More importantly, we showcase how developers can utilize AIRS to gain a deeper understanding of agent behavior, and debug specific DRL policies and even the application designs.

In particular, using AIRS, we successfully discovered two bugs in the DRL reward function in the Microsoft CyberBattleSim framework for lateral movement attack and defense [45]. We reported these bugs to Microsoft's 365 Defender Team who later acknowledged our findings. This provides empirical evidence that AIRS can be used to debug DRL-based security applications.

In summary, our paper has the following contributions:

- We propose AIRS, a general framework to explain deep reinforcement learning models in the security domain by providing the *step-level* explanation. AIRS is designed for high explanation fidelity, stability, and efficiency.
- To evaluate AIRS, we implement various baselines (including existing DRL explanation methods) and test them on four different security applications. We confirm AIRS outperforms existing approaches and baselines.
- We provide case studies to show how to use AIRS to debug DRL-based security applications. We have found real-world bugs using AIRS.

2 Background

2.1 Modeling an RL Problem

A sequential decision-making problem can be formalized as an RL learning task, in which an agent observes the task environment and takes proper actions to fulfill the task. Take autonomous driving as an example. The self-driving car (i.e., agent) observes the road condition and takes a series of proper driving actions (e.g., accelerating and turning) to reach a designated destination. In this process, the agent receives a reward after taking each action that measures how well it performs at each time step. The goal of RL is to learn an optimal policy for the agent. By taking actions given by the policy, the agent could collect a maximum amount of total reward, indicating it could fulfill the task in an optimal way. In the above autonomous driving example, this optimal way could refer to safely reaching the final destination in the shortest time. In the setting of DRL, the policy network is modeled as a deep neural network (i.e., a policy network). At each time step, this policy network takes as input the observation of the environment (e.g., the current snapshot of the road condition in the above example) and outputs the corresponding action that the agent would take. Solving a DRL problem is equivalent to learning the parameters of this policy network.

To learn the policy network, we first need to model a DRL problem as a Markov Decision Process (MDP), which is represented as a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$. In this tuple, $\mathcal{S}$ and $\mathcal{A}$ are the finite state and action sets, in which each state $s^{(t)}$ and action $a^{(t)}$ represents the state and action of the agent at time t. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition function, where $T^a_{ss'} = \mathbb{P}[s^{(t+1)} = s' | s^{(t)} = s, a^{(t)} = a]$ denotes the probability that the agent transits from state s to s' by taking action a at time t. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, where R^a_s represents the reward if the agent takes action a at state s.

As is mentioned above, the goal of DRL is to train a policy network $\pi(a|s)$ for the agent that maximizes the agent's total reward. Formally, the total reward can be represented by the *state-value function* $V_\pi(s)$ defined as

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(R^a_s + \gamma \sum_{s' \in \mathcal{S}} T^a_{ss'} V_\pi(s')), \quad (1)$$

or the *action-value function* $Q_\pi(s,a)$ defined as

$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} T_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a'), \quad (2)$$

where $\gamma \in [0,1]$ is a discount factor that controls the agent's focus on immediate rewards or long-term rewards. The state-value function $V_\pi(s)$ is the expected total reward of an agent starting from state s. Slightly different from $V_\pi(s)$, the action-value function $Q_\pi(s,a)$ is the expected total reward of the agent starting by taking action a at s. The value of both functions measures the quality of the agent's policy π. An optimal policy could be obtained by maximizing either function, ensuring that the agent receives the maximum rewards from the environment.

2.2 Learning a DRL Policy

Recent works have proposed many algorithms to solve the optimal policy, among which the most widely used ones are DQN [48] and PPO [57]. As mentioned above, these two algorithms are also widely adopted in DRL-based security applications. Below, we briefly introduce these algorithms.

Deep Q-learning (DQN). Deep Q-learning does not explicitly learn a policy network. Instead, it uses a deep neural network to approximate the action-value function introduced above, which takes as input state s and action a and outputs the estimated Q value. With this approximation, the agent follows a policy that suggests the action with the highest Q-value at each time step. As is shown in recent research, such a method demonstrates a great success in many applications with a discrete action space, such as Atari games [47] and the security applications of selfish mining, and the network lateral movement used later in our evaluation.

Proximal Policy Optimization (PPO). Different from DQN, PPO directly parameterizes the policy network as $\pi_\theta(s,a) = \mathbb{P}(a|s,\theta)$. At time t, this function takes as input state $s^{(t)}$ and outputs the corresponding action $a^{(t)}$. To learn this policy network, PPO proposes an objective function, which combines the state-value function in Equation (1) with the difference between an updated and an old policy. By maximizing this objective function, PPO could obtain a policy that collects the maximum amount of rewards. In addition, this objective function could enable the PPO to monotonically improve the agent's total reward during the training phase. With this property, the training process convergences faster and is more stable than previous training algorithms that learn policy networks directly. PPO is the state-of-art algorithm for DRL tasks with a continuous action space, such as the safe driving application introduced in Section 5.

3 DRL Explanation Methods and Limitations

Existing research on explaining DRL mainly focused on feature-level explanation. Technically speaking, these methods can be categorized into post-explainable methods and self-explainable techniques. Post-training explanation methods are originally designed to identify input features most critical to a DNN's prediction (e.g., white-box gradient-based methods [14,42], black-box approximation-based methods [20,21]). As introduced in Section 2, a policy network is also a DNN taking the current observation as input and outputting the corresponding action. As such, researchers extended post-training methods to explain policy network (e.g., [11,18,35,40,41,43,53,60,67,69]), identifying features in observation most important to an agent's action. Unlike post-training explanation methods, self-explainable techniques replace the non-explainable policy network with a self-explainable model. The self-explainable model unveils the association between observation and an agent's action and thus explains which features contribute the most to the agent's action (e.g., [32,49,50,66,81]). As discussed in Section 1, these methods cannot provide step-level explanations that draw insights into the final result of the agent.

To the best of our knowledge, only a few explanation methods [4,22,31] could provide step-level explanations. However, they are not suitable for explaining DRL-based security applications due to the following limitations. First, value function-based methods [4, 31] utilize the value function to assess state importance. As mentioned in Section 2, value function captures the relationship between states and the agent's expected total reward rather than the final rewards in individual episodes. Due to this misalignment, using value function to explain individual episodes cannot give a high explanation fidelity (cf. Section 6). Besides, in some real-world cases, value function is not released together with the policy network, since it does not guide the agent's action, and value function-based methods cannot be applied to these cases. Second, a recent method, EDGE [22], focuses on providing coarse-grained global insights within a set of collected episodes. More specifically, EDGE [22] utilizes an interpretable model to predict the final reward of the collected episodes and thus capture the global importance of actions/states to a game's final result. As is discussed in Section 1, many security applications need explanations that go beyond global insights. They need to identify the important actions/states of each individual episode. With this ability, as is showcased in Section 7, security developers could utilize such explanations to improve their DRL-driven security applications. Third, some methods (*e.g.* EDGE) employ a relatively sophisticated model, introducing complicated computation and optimization. When porting it to security applications, the complexity of the applications introduces even more challenging engineering tasks to these methods. To this end, we introduce a novel DRL explainable framework to derive an explanation for each individual episode without introducing overly sophisticated computation and optimization.

4 Proposed Technique

In this section, we first describe the research problem we aim to address and the assumptions we make. Then, we discuss the technical challenges confronted when addressing the problem (i.e., explaining deep reinforcement learning). After that, we present how we handle the challenges and design our explanation method, followed by an overall algorithm.

4.1 Research Problem & Assumptions

Many security applications learn a deep policy by using DRL algorithms (e.g., Q-learning [26, 28, 47, 70, 74] or policy gradient [23, 46, 56, 57]). Then, they utilize the policy to take a series of actions and thus generate the desired final reward for that application. Take malware mutation as an example. To generate a malware variant that could bypass a target detector, security researchers recently utilized a DRL algorithm to learn a deep policy. The policy takes as input a malware sample and outputs a sequence of actions indicating how to mutate that malware step by step and thus obtain a valid malware variant.

In this work, our goal is to pinpoint the agent's actions most critical to the (un)desired reward in single-agent RL environments that can be modeled as a Markov Decision Process (MDP). As discussed in Section 2 and 5, this is the most widely adopted RL model in DRL-based security applications. Take the malware mutation as an example. We aim to identify the actions that contribute the most to the failure/success of malware variant generation. As we will showcase in Section 7, this ability could help security researchers establish trust in their DRL model, offset unexpected model outcomes, and even track down the design flaw of the DRL model.

Before discussing our proposed DRL explanation method, we make the following assumptions. First, we assume that our explanation method could access the input to the deep policy network (i.e., the observation of the environment) and the final reward gained through the policy. Take the malware mutation as an example. We assume the access to the malware sample and the reward that the deep policy could obtain from the sequence of mutation actions. Second, we do not assume access to the value/Q function or the parameters of the policy network. This assumption ensures that our explanation method treats the deep policy network as a black box.

4.2 Basic Idea & Challenges

As is mentioned in Section 2, after taking each action, the agent turns its current state into a new state. Take the malware mutation as an example. After taking a mutation action, the malware sample is converted into a new variant, which will then be further converted into another new variant after one utilizes the policy network to take the following mutation action. As a result, we can treat the DRL explanation as a task that finds the states most critical towards the final reward.

To do it, one basic idea is to model the relationship between the sequence of states and the final reward. For example, we can model the relationship in a linear form:

$$r = \sum_{i=0}^{T} \theta_i \cdot g_{\text{PCA}}(\mathbf{s_i}) + b \tag{3}$$

in which r and $\mathbf{s_i}$ are the final reward and the state at the i-th time step, respectively. $g_{\text{PCA}}(\cdot)$ is a dimensionality reduction function, Principal Component Analysis [15], that converts the vectorized state $\mathbf{s_i}$ into a singular value so that, through a linear combination parameterized by $\theta_0, \theta_1, \cdots, \theta_T$, one could predict the final reward r. Because the linear relationship is self-explainable, we can treat the states with the parameters in the highest values as the most critical toward the final reward.

While this method is intuitive, the explanation derived from this method might not be of high fidelity.

- The state transition sequence $\mathbf{s_0} \cdots \mathbf{s_T}$ in deep reinforcement learning is time-dependent. Without considering the temporal relationship between states, a simple linear combination would inevitably introduce modeling errors.
- To learn the parameters of the linear model, we need to gather many episodes (i.e., state transition sequences and the corresponding final rewards). However, the parameters $\theta_0 \cdots \theta_T$ learned from these episodes do not reflect the state importance in a single episode but the importance in all episodes used in the model learning. As a result, given an individual episode, the method above cannot pinpoint which state in an individual episode is more critical to the corresponding final reward.
- PCA conducts a simple linear transformation to reduce dimensionality. Prior research [2, 17] has demonstrated that PCA may not be a good option, especially when the original dimension is high and the converted dimensionality is low. In such cases, PCA cannot preserve the majority of critical information in original inputs and thus further impacts model fitting and the explanation fidelity.

4.3 Our Method

To tackle the three challenges above, we improve the intuitive method above from three perspectives. To capture the temporal dependency of the state transition, we first pass the state sequence to a recurrent neural network. As is depicted in Figure 2, the recurrent neural network $H(\cdot)$ is a sequence-to-sequence model. It not only converts state sequence into a new sequence but, more importantly, ensures that the element $H(\mathbf{s_i}) = \mathbf{x_i}$ in the new sequence $(\mathbf{x_0}, \mathbf{x_1}, \cdots, \mathbf{x_T})$ captures the time dependency of previous states $\mathbf{s_0}, \mathbf{s_1}, \cdots, \mathbf{s_{i-1}}$.

While the introduction of a recurrent neural network resolves the time-dependency issue, the proposed method still suffers from the other two challenges above. To address the two other challenges, as is shown in Figure 2, we first replace

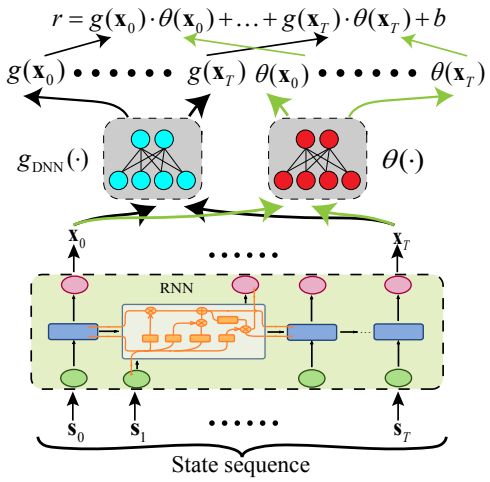


Figure 2: **Approximation Model of** AIRS—It takes the state sequence as input, and predicts the corresponding reward as output.

the PCA with a fully connected deep neural network $g_{\text{DNN}}(\cdot)$ to further process the temporal encoded $\mathbf{x_i}$. Unlike PCA that only capture linear relationships, this deep neural network could model the complex and non-linear correlations within in $\mathbf{x_i}$. As discussed in existing works [2], DNN is much better than PCA in preserving critical information when performing dimensional reduction for high-dimensional inputs.

Finally, to tackle the last challenge (i.e., linear regression cannot provide an explanation of each individual episode), we further improve the method above by replacing the linear regression coefficient $\theta_i \in \{\theta_0, \theta_1, \cdots, \theta_T\}$ with a deep neural network $\theta(\cdot)$. As we can see from the equation below:

$$r = f(\mathbf{x_0}, \mathbf{x_1}, \cdots \mathbf{x_T}) = \sum_{i=0}^{T} \theta(\mathbf{x_i}) \cdot g_{\text{DNN}}(\mathbf{x_i}) + b \qquad (4)$$

the neural network $\theta(\cdot)$ takes $\mathbf{x_i}$ as input. As is depicted in Figure 2, for each episode $\mathbf{s_0}, \mathbf{s_1}, \cdots, \mathbf{s_T}$, it outputs a unique vector $\theta(\mathbf{x_i})$. By aggregating the multiplication of $\theta(\mathbf{x_i})$ and $g_{\text{DNN}}(\mathbf{x_i})$, we could obtain the prediction of the final reward r. Using this design, we address the problem of not being able to derive an explanation for each individual episode simply because for each episode, $\theta(\cdot)$ outputs a sequence of distinctive coefficients indicating the importance of the states towards the final reward.

4.4 Other Critical Design

In this work, we can gather various episodes and utilize these episodes to learn the parameters in Equation (4). However, there are still a couple of issues that need to be taken care of.

Handling stability of explainability. Our proposed method may encounter an instability issue. When an episode is similar to another, we would like to have an explanation similar to each other. However, by solving the equation above, we may not be able to obtain such a property. As a result, we introduce a constraint below.

$$\begin{aligned} &\left\|\theta(\mathbf{X}^{(i)}) - \theta(\mathbf{X}^{(j)})\right\| < L\left\|\mathbf{X}^{(i)} - \mathbf{X}^{(j)}\right\| \\ &\text{for every } \mathbf{X}^{(i)} : \left\|\mathbf{X}^{(i)} - \mathbf{X}^{(j)}\right\| < \delta \end{aligned} \qquad (5)$$

This constraint bounds the variation of explanation when the episodes $\mathbf{X}^{(i)}$ and $\mathbf{X}^{(j)}$ are similar. Here, $\left\|\mathbf{X}^{(i)} - \mathbf{X}^{(j)}\right\|$ indicate the similarity of two episodes. $\left\|\theta(\mathbf{X}^{(i)}) - \theta(\mathbf{X}^{(j)})\right\|$ represents the similarity of the states' importance. The constraint above implies that their corresponding states' importance should also be similar if the episodes are similar.

To facilitate the computation when optimizing the parameters in our proposed method above, we further transfer the constraint as a regularization term below.

$$\mathcal{L}_\theta(f(\mathbf{X}^{(i)})) := \left\|\nabla_X f(\mathbf{X}^{(i)}) - \theta(\mathbf{X}^{(i)})^\top \mathbf{J}_X^g(\mathbf{X}^{(i)})\right\| \approx 0 \qquad (6)$$

Here, $\mathbf{J}_X^g$ is the Jacobian of g_{DNN}. This regularization term also forces the model to act as a linear model locally [3].

Filtering out commonly non-important states. After we learn the approximation model by using many episodes Γ gathered from the target agent, we can then use $\theta(\mathbf{X}^{(i)})$ to derive the explanation for each episode. However, before choosing the critical states by using $\theta(\mathbf{X}^{(i)})$, we further filter out those states that commonly appear in all episodes. Those states are non-critical to the final reward. If not filtering out at this stage, they may introduce noise to our explanation.

To this end, we further introduce a filtering mechanism. This filtering scheme first computes the correlation between the final reward and each state as follows

$$s(t) = corr(\mathbf{s}_i, \mathbf{r}), \qquad (7)$$

where $\mathbf{r}$ represents the batch of $r^{(k)}$ $(k = 1, 2, \cdots, N)$. If the correlation between the final reward and the state is low, it indicates that the state commonly appears in most episodes and has no impact on the final reward. Therefore, we do not consider those least correlated states in our explanation.

4.5 Proposed Explanation Algorithm

Finally, we show the proposed explanation algorithm. As shown in Algorithm 1, we first initialize the parameters of three networks (Line 2). Then, we input the training episodes to the RNN feature extractor H and obtain a latent representation X, which captures the temporal dependency between states (Line 4). With the latent representation, we then feed it into g_{DNN} and θ to predict the final reward (Line 5). Next, we compute the training loss, which is composed of the mean square error of the predicted reward and the stability regularization in Eqn. (6) (Line 6-7). Finally, we conduct an end-to-end update of the parameters in the three networks to

Algorithm 1 Explanation model training of AIRS.

1: **Input:** training episodes $\Gamma = (\mathbf{S}, \mathbf{r})$, training epoch K, constant λ
2: Initialize $H(\cdot), \theta(\cdot)$ and $g(\cdot)$
3: **for** `k=1,2,...,K` **do**
4: Compute hidden representation $\mathbf{X} = H(\mathbf{S})$
5: Compute the predicted final reward based on Eqn. (4)
6: Compute the mean square error loss $\mathcal{L}_{mse} = \frac{1}{N}\mathbf{e}^\top \mathbf{e}$, where $\mathbf{e} = f(\mathbf{X}) - \mathbf{r}$
7: Compute the final loss $\mathcal{L} = \mathcal{L}_{mse} + \lambda \mathcal{L}_\theta$, where $\mathcal{L}_\theta(f(\mathbf{X}))$ is the regularization term in Eqn. (6).
8: Update $H(\cdot), \theta(\cdot)$ and $g(\cdot)$ to minimize $\mathcal{L}$
9: **end for**
10: Compute step filter m
11: **Output:** trained model $H(\cdot), \theta(\cdot)$, step filter m

Algorithm 2 Explanation generation of AIRS.

1: **Input:** Testing episode $\mathbf{S}^{(i)}$, $H(\cdot), \theta(\cdot)$, and m
2: $\theta(\mathbf{X}^{(i)}) = \theta(H(\mathbf{S}^{(i)}))$
3: **for** `t=1,2,...,T` **do**
4: $v_t = \begin{cases} |\theta(\mathbf{X}^{(i)})_t|, & \text{if } t \in m \\ 0 & \text{if } t \notin m \end{cases}$
5: **end for**
6: **Output:** importance score $\mathbf{v} = (v_1, v_2, \cdots, v_T)$

minimize the training loss with a gradient-based optimization method (Line 8). After training the networks, we further compute the step filter in line 10.

As depicted in Algorithm 2, given a new episode $\mathbf{S}^{(i)}$, we use $\theta(H(\mathbf{S}^{(i)}))$ to assess its initial time-step importance (Line 2). Then, we filter out the non-important states using the step filter m (Line 4) and derive the step importance score for this episode.

5 Representative Applications

In recent years, we have witnessed applications of deep reinforcement learning (DRL) in various security-sensitive domains. Below, we introduce a set of representative tasks. Then, in the next section, we assess the performance of our proposed explanation method in such tasks.

Autonomous Driving. We first consider the application of DRL in improving driving safety (e.g., [36, 38, 51, 52, 73]). We use the state-of-the-art autonomous driving framework MetaDrive [37] as a concrete example. It simulates a driving environment where a DRL algorithm learns a deep policy network to navigate a vehicle to a destination in a safe and fast manner. Specifically, MetaDrive converts the Birds Eye View (BEV) of the road condition (as illustrated in Figure 3) as well as the driving vehicle's surrounding information into a vector representation, which encodes the vehicle's current

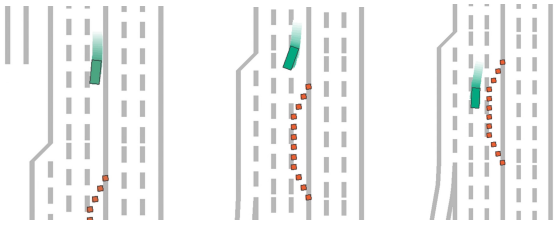


Figure 3: **BEV of Safe Driving**—The green rectangle represents the car controlled by the DRL agent. Its goal is to drive along the road and avoid road barriers indicated by red squares.

state, including its steering, direction, velocity, and relative distance to traffic lanes. The policy network takes as input this state vector and outputs the driving action (i.e., the vehicle's acceleration, brake, and steering). MetaDrive designs a set of reward functions. For example, when the driving agent has a collision with obstacles or violates traffic rules, MetaDrive assigns a penalty to the policy; on the contrary, the agent receives a reward when the vehicle arrives at the destination without any penalty. To prevent the vehicle from moving slowly or driving in a zig-zag way, MetaDrive also designs rewards to encourage the vehicle to move forward and maintain a reasonable speed.

Malware Mutation. In both academia and industry, deep learning has been widely adopted to detect Android malware [9, 44, 78, 79, 82]. To evade such detection, recent research has proposed HRAT, a DRL-based malware mutation method [83]. Unlike the conventional methods that employ adversarial learning for malware mutation (e.g., [19]), HRAT relies on a policy network that takes as input the call graph of the target malware and outputs the corresponding action to guide the malware manipulation.

Specifically, the action space of HRAT includes "adding function call", "rewiring function call", "inserting methods" and "deleting methods". As these actions do not vary the semantics of the malware, each mutation action generates a new valid malware variant. Based on whether the variant circumvents the target malware detector, HRAT assigns a reward or a penalty to the policy network and updates its parameters accordingly.

Selfish Mining in Blockchain. Selfish mining [13] is a deceptive cryptocurrency mining practice in which an unlawful miner withholds its newly mined blocks from the public blockchain. This action creates a fork, which is mined ahead of the public blockchain. The unlawful miner can then introduce this fork to the network, overwrite the original blockchain, and steal cryptocurrency from other users.

However, the strategy of determining whether and when to reveal the fork may significantly influence the reward that the unlawful miner eventually receives. For example, when an unlawful miner withholds many newly mined blocks, but the fork is only slightly ahead of the public chain, the decision to continue withholding mined blocks for a more extended

period or reveal the fork to the network immediately would significantly impact the miner's reward.

To find the best strategy that maximizes a selfish miner's reward, recent research [29] has proposed using DRL to learn a mining policy. The policy network takes the length of the current public chain and the fork as input and outputs the action that an unlawful miner should take. In [55], four unique actions are defined for an unlawful miner, which include "waiting", "overriding", "matching", and "adopting" (their concrete meanings deferred to Appendix A). Leveraging this policy network, the unlawful miner is able to gain a higher mining reward than following a naive selfish mining strategy.

Network Lateral Movement. Network lateral movement [8, 12, 68] describes the process in which an attacker breaches an enterprise network by exploiting the vulnerabilities to move from one network node to another. For example, an attacker often does not have all of an organization's sensitive data accessible from their machine; thus, the attacker needs to move to other machines to access the sensitive data. Previous works [10, 24, 45, 58] studying lateral movement usually simulate an enterprise network as a set of network nodes connected in a particular topology (pre-defined by the simulation user). Besides, they also pre-define a set of vulnerabilities that the attacker may exploit to move from one node to another. Meanwhile, a defender monitors the network activity to detect the attacker's presence and further prevent the attack. The attacker's goal is to take ownership of the enterprise network without being caught by the defender.

To study how efficiently the attacker is able to achieve this goal, recent research has proposed using DRL to learn a policy network to represent the optimal attack strategy [45]. Starting from one infected node, the attacker consults with the policy network to determine the next target node and the vulnerability to exploit; after breaching the next node, the attacker repeats this procedure until partially or fully taking control of the network. To train the policy network, a reward is assigned after each successful breach. The reward is a floating number representing the intrinsic value of the breached node (e.g., a SQL server has a greater value than a test machine). In addition, taking ownership of the whole network can bring a huge positive reward.

Figure 4 illustrates the procedure above. Given a compromised Windows 7 node under the attacker's control, the policy network takes as input the partially observable network (i.e., the nodes directly connected to the compromised node) and outputs an action the attacker is suggested to take on the compromised node (e.g., exploiting the vulnerability in the SMB file-sharing protocol). After taking the action, the attacker successfully moves on to a Windows 8 node. From there the attacker further follows the suggested action of using cached credentials to sign in to another Windows 7 machine and exploits the IIS remote vulnerability to own the IIS server.

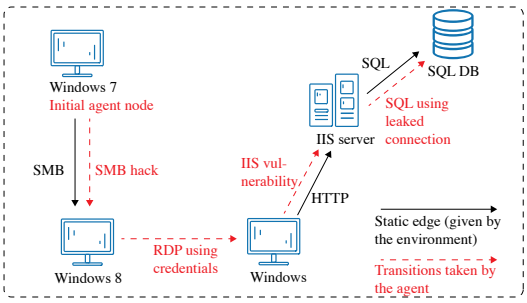


Figure 4: **Network Lateral Movement**—Example of lateral movement in a network.

6 Experiments

In this section, we conduct an empirical evaluation of AIRS in the four representative tasks: selfish mining [29, 55], lateral movement, autonomous driving, and malware mutation. We begin by introducing the experimental setting. We choose the CyberBattleSim [45] released by Microsoft to simulate the lateral movement environment and pick two environments with different topologies: chain-graph and random-graph.

6.1 Experimental Setting

6.1.1 Baseline Methods

Recall that our goal is to derive explanations that connect individual steps and the final reward (cf. Section 3). To this end, with episodes sampled from the agent, we may train an explainable approximation model to predict the reward and derive explanations from this model (like AIRS). For this approach, we choose to append a self-explainable attention module [6] or a linear regression (LR) module to the RNN feature extractor as our baselines.[1] After we train the explainable model on the collected episodes, we directly derive explanations from it. Comparing AIRS with these two methods evaluates the effectiveness of our explanation model design. Alternatively, we may use a non-explainable approximation model to fit the episodes and combine it with a post-training method to obtain the explanations. Here, we use a typical RNN model to fit the episodes and select three widely used post-training gradient-based methods, Vanilla gradient [63], Integrated gradient [65], and Smooth gradient [64], to explain the trained RNN model. In addition, when value functions are available, one could also use the state/action value function to determine the state importance. As such, we also include two value function-based explanation methods, Highlights [4], and Trust [31] as our baseline approaches. Due to the space limit, we put some experimental details in Appendix B.

[1] Note that LR gives the global explanation and we apply the same explanation for all traces.

6.1.2 Evaluation Criteria

We follow [75] to use three criteria to evaluate AIRS and baseline methods: fidelity, stability, and efficiency.

Fidelity: Recall that in this work we focus on step-level explanation, which identifies the steps with the greatest influence on the final reward (i.e., the critical steps). The fidelity of explanation measures the accuracy of the identified critical steps. In selfish mining and network lateral movement, the episodes are relatively short. We thus only focus on the most important single step, which is computed by the importance score $\theta(\mathbf{X})$ within the selected sub-sequence. In safe driving and malware mutation, the episodes are fairly long while a single step does not have an obvious influence on the final reward. We thus attempt to identify the most critical sub-sequences. We use a sliding window to step through all the filtered time steps, and then choose the sub-sequence with the highest average importance score. The width of the sliding window is 5% of the episode length. After identifying the most important step(s), we use the metric of *Relative Reward Difference* (RRD) to evaluate the fidelity of the explanations. We replace the action(s) at the selected time step(s) with random action(s) and then measure the average reward change. We use R_{origin} and R_{select} to denote the reward before and after the action manipulation. Due to the varying reward design in different applications, the magnitude of reward change may vary. Thus, we use random selection to normalize the reward change for each application: we randomly select the step(s) and perform the manipulation, with the reward change denoted by R_{random}. Formally, RRD is defined as:

$$\text{RRD} = \frac{|R_{select} - R_{origin}|}{|R_{random} - R_{origin}|} \tag{8}$$

Here we use the absolute value because we would like to concentrate on the time step that makes the greatest contribution towards the final reward, regardless of positive or negative contribution. In other words, the critical time step can lead to a huge reward or incur a great penalty. We test 500 episodes for each application to calculate the average RRD. We repeat this experiment 10 runs and compute the mean and standard deviation of the resulted average RRDs. Note that a greater difference represents a better explanation. If the reward difference is similar or even smaller than the difference with respect to random selection, then the explanation is ineffective.

Stability: In addition to high fidelity, the explanations generated in security-sensitive applications also need to be stable. That is, the chosen critical steps should remain the same across different runs in order to be useful. The possible fluctuations may come from two sources: the randomness of model initialization and the randomness of explanation calculation. For the first source, the random parameter initialization of the underlying deep neural network model may result in different explanation results provided by the model. For the second source, for example, the Integrated gradient and Smooth gradient introduce additional randomness by selecting the noise [64] or baseline [65] randomly during computing the explanation. Both sources of randomness may influence the stability of the generated explanations. To measure the interpretation stability, for each interpretation method, we generate explanations in 10 runs with different random seeds and measure the average L_1 distance of the results between each pair of runs. For each application, we normalize the distance by the length of the episode.

Efficiency: An efficient method should take a reasonable training time and should be able to provide a real-time explanation (during testing time) as the DRL agent is running. Thus, we evaluate the efficiency from two aspects. First, we compare the training time of explanation methods that requires to train an approximation model (other than Highlights and Trust). Then, we measure the time cost of generating explanations for 500 episodes. In both experiments, we measure the average time cost in 10 runs.

6.2 Experiment Results

Fidelity Evaluation. Figure 5 compares the explanation fidelity of AIRS and alternative methods in different applications. We have the following observations.

First, AIRS achieves the highest RRD in all applications, showing its superior fidelity. Recall that RRD measures the reward difference after manipulating the critical time step, the higher RRD shows the higher importance of selected steps since replacing them with random actions could bring a great change in the final reward. This verifies the effectiveness of our explanation methods.

Second, compared with AIRS, other model-based methods all have lower fidelity scores, while the gaps vary across different applications, showing much larger margins in chain-graph and safe driving than selfish mining and malware mutation. This may be explained by the episode lengths in different applications. Recall that the episode length of selfish mining is the shortest and the episode lengths of safe driving and malware mutation are rather long. For long episodes, one needs to derive importance scores from long outputs of the RNN model. Simple linear regression or a single attention module may not work well when the output is long [25], while gradient-based methods suffer from the saliency vanishing issue [33, 34] for long outputs. Meanwhile, AIRS uses a deep neural network $\theta(\cdot)$ to process the RNN outputs and is able to extract the importance score more precisely.

Third, similar to the model-based baseline approaches, value function-based methods also have lower fidelity scores than AIRS. As discussed in Section 3, this is because value functions model the contributions of states to the agent's expected total return rather than the associations between states and the final reward in each episode. As such, they cannot provide highly faithful explanations to individual episodes.

Finally, we find that although AIRS has the highest RRD

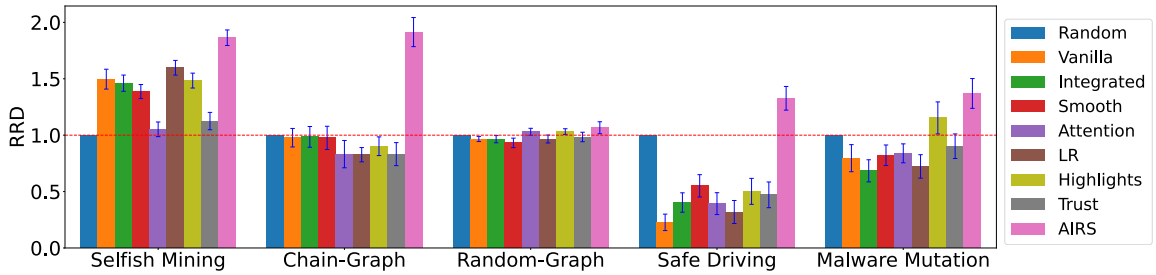


Figure 5: **Fidelity Evaluation Results**—"Chain-Graph" and "Random-Graph" represent two different topology setups for the lateral movement application [45]. A higher RRD score indicates better fidelity of the explanation. The horizontal dotted line represents RRD=1. If the RRD score is lower than 1, the explanation method would fail to pinpoint the critical time step in that application. The error bar in the figure shows the standard deviation.

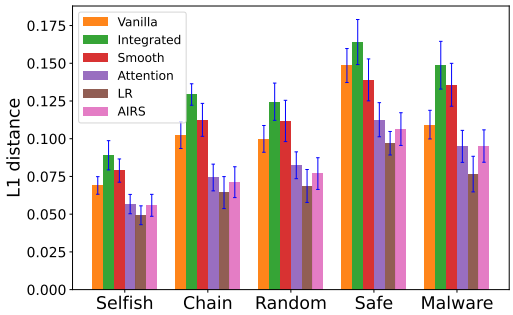


Figure 6: **Stability Evaluation Results**—We show the average L_1 distance in each pair of run and the standard deviation. LR, as a global explanation method, has the L_1 lowest distance which means LR is the most stable method, followed by AIRS.

in the Random-Graph application, the gap is relatively small compared with other applications. We list the RRD and reward difference in Table 6 for a more detailed comparison. it is observed that while the advantage of AIRS in RRD is marginal over other methods, its reward difference is much higher than alternative methods, indicating that AIRS is able to identify the critical steps more precisely.

Stability Evaluation. We show the stability evaluation results in Figure 6.[2] We have the following observations.

First, LR has the highest stability (i.e., the lowest average pair-wise L_1 distance between different runs) among model-based methods. This is expected because LR offers a global explanation: if the weights of linear regression across different runs are similar, the given explanations remain consistent.

Second, AIRS and the attention method share the highest stability next to LR. This can be explained by that in both methods, once the approximation model is fit to the collected data, the explanation is derived by feed forwarding the model. Thus, the fluctuation is only raised by the randomness of the model training. We also observe that AIRS is slightly more stable than the attention method. This is attributed to the stable regularization term described in Equation (6).

Finally, among all the methods, the Integrated gradient and Smooth gradient show the lowest stability, which is especially evident in safe driving and malware mutation with a long episode. This may be explained by that for gradient-based methods, the instability not only comes from the randomness during model training but is also raised by the randomness of computing explanations after model training (e.g., Smooth gradient needs to randomly add noise while Integrated gradient needs to randomly choose the baseline). Through the experimental results, we verify that AIRS has high stability.

Efficiency Evaluation. The results of the efficiency evaluation are shown in Table 1. As we can first observe from the table, AIRS has a comparable training time with other model-based methods in all applications. This verifies that our model design does not introduce additional training cost over existing approximation models. Regarding the generation cost, as also shown in Table 1, AIRS is much faster than most baseline method except Vanilla gradient and Attention. Specifically, value function-based methods have much higher generation time cost on most applications. This is because they need to traverse the entire episode to compute the value for each state. For games with a long episode, these method are expensive. Regarding Integrated gradient and Smooth gradient, they are slow because of the need to perform iterative expensive back-propagation to calculate the average gradient. Through the experimental results, we can verify that AIRS has high training and generation efficiency.

In addition to the above experiments, we also compare AIRS with a naive solution that exhaustively randomizes the agent's action at each state to assess state importance. As detailed in Appendix C, AIRS significantly outperforms this method in explanation efficiency.

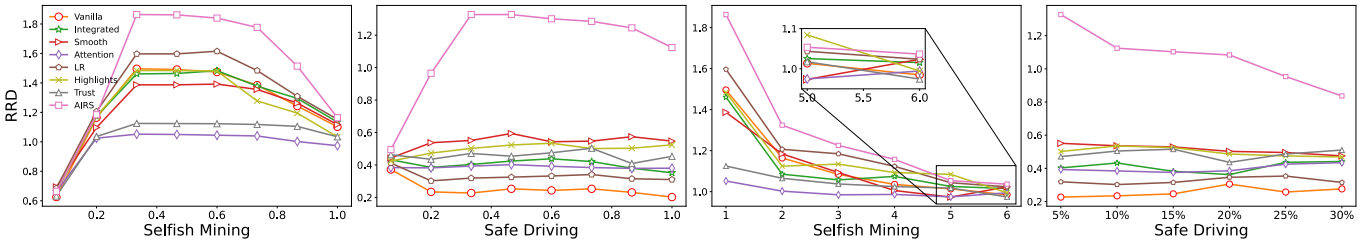
6.3 Ablation Study

We conduct an ablation study on selfish mining and safe driving to investigate how the hyperparameters influence the performance of these explanation methods.

[2] Note that value function-based methods do not introduce any randomness and thus always have zero L_1 distances. We do not show them in the figure.

Method	Selfish Mining			Chain-Graph			Random-Graph			Safe Driving			Malware Mutation		
	Train(s)	Generate(s)	Total(s)	Train(s)	Generate(s)	Total(s)	Train(s)	Generate(s)	Total(s)	Train(s)	Generate(s)	Total(s)	Train(s)	Generate(s)	Total(s)
Vanilla	103.46	32.42	135.88	156.45	34.67	191.12	406.95	35.34	442.29	732.83	42.69	775.52	193.19	40.33	233.52
Integrated	103.46	160.38	263.84	156.45	167.90	324.35	406.95	180.39	587.34	732.83	261.65	994.48	193.19	250.67	443.86
Smooth	103.46	185.25	288.71	156.45	189.45	345.90	406.95	192.32	599.27	732.83	280.37	1013.20	193.19	276.40	469.59
Attention	158.08	24.11	182.19	186.49	22.16	208.65	433.23	23.27	456.50	768.84	26.34	795.18	204.17	24.74	228.91
LR	105.94	-	105.94	155.73	-	155.73	402.54	-	402.54	742.56	-	742.56	196.43	-	196.43
Highlights	–	56.98	56.98	–	389.69	389.69	–	798.52	798.52	–	1053.63	1053.63	–	321.59	321.59
Trust	–	57.35	57.35	–	385.88	385.88	–	783.75	783.75	–	1059.82	1059.82	–	320.34	320.34
AIRS	125.58	18.36	143.94	173.64	20.43	194.07	423.59	24.76	448.35	764.69	25.94	790.63	200.33	24.87	225.20

Table 1: **Efficiency Evaluation Results**—We show the average time cost of training approximation models and generating explanations. Value-based methods do not have the training time cost. Three gradient-based methods share the same approximation model and thus have the same training cost. LR gives the global explanation and once the regression model is trained it has no time cost to generate explanations.



(a) vary length of filtered subsequence (b) vary number of perturbed steps

Figure 7: **Ablation Study Results**—In (a), we vary the length of the filtered sub-sequence and record the RRD scores. When the length is close to 0, most of the states are filtered out and only a few states are left. When the length is approaching 100% of that before filtering, few states are filtered out. When it is exactly 100%, there is no filter. In (b), we vary the number of selected critical steps and perturb them.

First, we study the relationship between the fidelity and the length of the filtered sub-sequence. As discussed in Section 4.4, we use a step filter to filter out the common non-important states before identifying the critical steps. Here, we vary the length of this sub-sequence and record the RRD scores in Figure 7. For selfish mining, when the length is in a reasonable range (e.g. between $1/3$ and $2/3$), the RRD scores remain high and is not very sensitive to the change in the length. In this case, the explanation methods perform well and have high fidelity. However, when the length is approaching 0, the RRD scores drop quickly below 1. This is because most of the states are filtered out, including those important states. It also shows that using correlation alone cannot capture the critical step well. Finally, we can find that as the length is increased to 100%, the RRD scores also drop quickly for all explanation methods. In this condition, no pre-selection is used and the explanation performance is degraded. In safe driving, only AIRS shows a similar trend, whereas other methods have little variations under different lengths. This indicates they cannot correctly select the critical steps with or without the step filter.

Second, we would like to analyze the performance when more steps are identified. For selfish mining, we select the most important sub-sequence with one to six steps following the sub-sequence identification for safe driving and malware mutation. For safe driving, we perturb up to 30% steps. As we can observe from Figure 7, generally, when the number of perturbed steps is increasing, RRD scores drop quickly. When perturbing up to 6 (or 30%) steps, RRD scores of all explanation methods are below 1 for both applications, which means they are ineffective. Although AIRS remains the highest RRD score, it also fails in this case. One possible reason is raised by the sensitivity of such applications. They can not tolerate random actions in a long sub-sequence, which will lead to early termination, regardless of whether the sub-sequence is critical or not. Thus the explanation for those security applications should focus on the single-step or a short sequence, which is also helpful for expert analysis.

7 Utility of Interpretation

In this section, we discuss the utility of the interpretation generated by AIRS. With a series of case studies, we illustrate how the generated explanation can help with security applications that depend on deep reinforcement learning (DRL). We group the case studies into three main utility scenarios:

- **Agent Behavior Understanding.** The goal is to provide a deeper understanding of the agent behaviors using the explanation results. By highlighting the critical steps in a long sequence of actions, AIRS can help developers to examine whether the agent behavior follows their intuitions or design goals.
- **Policy Debugging.** Through explanation, developers can debug the errors of the policy network, and potentially improve the performance of the DRL agent by patching

	Episode Reward	Cost	Success Rate
Before Retrain	278.94	0.1	0.8
After Retrain	298.90	0.0	1.0
Diff.	7.15% ↑	100% ↓	25% ↑

Table 2: **Retrain Results**—Retrain results in safe driving after interpretation.

problematic states.

- **Application Debugging.** When the generated explanation contradicts with human understanding, it may indicate bugs at the application level (e.g., the design of the DRL agent). This could be more serious compared with specific problems in the learned policies.

In the following, we present case studies for different applications, prioritizing the interesting and unexpected results.

7.1 Agent Behavior Understanding

We use safe driving and network lateral movement to demonstrate how explanations can help developers better understand the model behavior. Figure 8 shows two examples. The color bar indicates the importance score (the importance score increases as the color turns from yellow to red).

Safe Driving. Figure 8(a) shows one example to explain why a particular driving episode is successful. In this example, AIRS highlights a few time steps when the vehicle is passing by traffic barricades and deems them as the most important steps. This explanation matches with human intuition as the success of the episode can be attributed to the success of avoiding collisions when passing by these traffic barricades. Hypothetically, if the vehicle were to collide with the obstacles, it will not only get the collision penalty, but also lose the arrival reward even if it can arrive at the destination. Overall, the explanation shows the expected behaviors of the agent.

Lateral Movement. Figure 8(b) shows an example of the lateral movement against a random-graph. It illustrates how the agent discovers and connects to the nodes in the network. The red nodes are controlled by the attacker and the blue nodes are newly discovered nodes at a given time. The solid arrows represent existing successful connections and the dashed arrow represents discovering new nodes. The explanation generated by AIRS indicates the last two time step is the most critical steps. Recall that in the reward design, taking ownership of the whole network can bring a huge positive reward. With this in mind, this interpretation conforms to the application design. The DRL agent tries to take ownership of the whole network to receive this huge reward.

7.2 Policy Debugging.

Next, we present case studies for safe driving and selfish blockchain mining to perform policy debugging with AIRS.

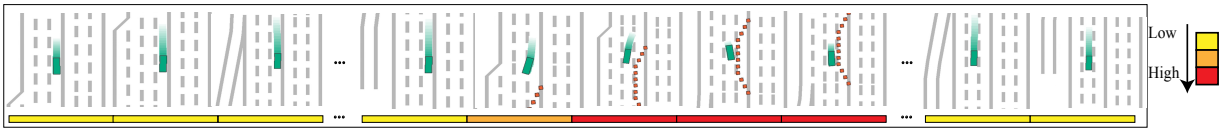
State ID	State	Patch Result	Positive Impact
1	[2,1,2]	+0.187	**Yes**
2	[3,3,2]	-0.443	No
3	[4,3,2]	+0.271	**Yes**
4	[4,3,0]	-0.240	No
5	[3,2,2]	-0.064	No
Final (Patch states 1 and 3)		**+0.301**	**Yes**

Table 3: **Patch results**—Patch results in selfish mining without retraining. The patch result is the average reward change compared to the original average reward.

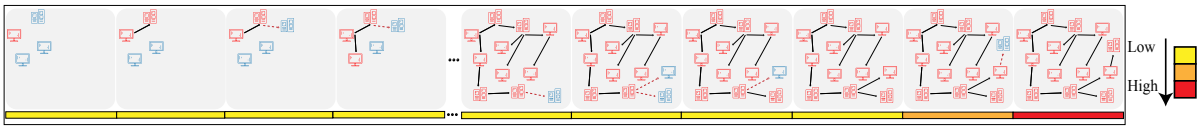
Safe Driving. For safe driving, we try to use AIRS to improve the learned policies. The high-level idea is to use AIRS to identify critical time steps and only re-train the model on these critical time steps. The intuition is that if the agent can perform better during these critical time steps, they should have a much better overall performance.

More specifically, we replay the collected episodes and sample the most important time steps identified by AIRS to form a dataset for retraining. We test the retrained agent for 500 episodes and show the average result in Table 2. The average episode reward is the sum of the driving reward, and the average cost means the driving penalty for collisions and violations of traffic rules. The success rate is the ratio of successful runs (where the vehicle arrives at the destination without collision) out of 500 runs. We observe that before retraining, the agent has a small probability to receive a driving penalty and fail to achieve the goal. However, after retraining on the selected critical time steps, the agent can drive more safely without collision or traffic rule violation. The success rate becomes 100% and the mean episode reward is much higher than before. The result confirms the utility value of retraining over selected critical time steps.

Selfish Mining. Different from safe driving, we show that we can utilize the explanation to debug the agent directly *without retraining the model*. The high-level idea is to analyze the critical steps and then introduce hard-coded rules to patch the errors of the agent. Table 3 lists 5 most frequently selected states by AIRS, sorted by the frequency (e.g., state $[2,1,2]$ has the highest frequency). While these states are considered important, they can either have a positive influence or a negative influence on the final result. To improve the agent, we first examine these states one by one to determine the direction of the influence if we patch it. More specifically, for each state listed in the table, we run a simple experiment. When the agent sees this given state, instead of taking the action computed by the policy network, we let the agent take a random action that different from the agents original action. When the agent sees other states, it still takes the actions suggested by the policy network. We record the average reward and compare it with the original reward. The reward change is reported in Table 3. As shown in the table, patching the action for state $[2,1,2]$

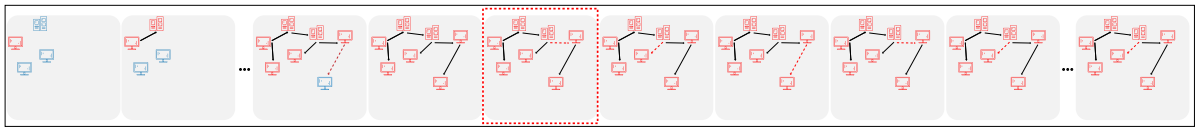


(a) **Safe Driving**: the green rectangle represents the car controlled by the DRL agent and the red squares represent the obstacles on the road

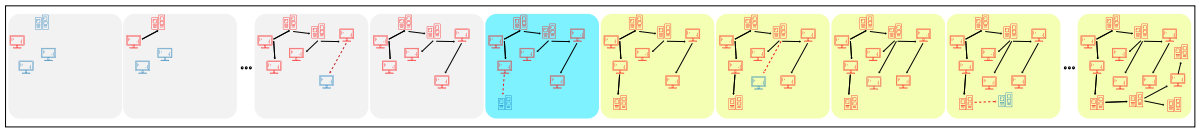


(b) **Lateral movement**: The attacker starts with one node and makes lateral movements. The red nodes are compromised nodes and the blue nodes are newly discovered nodes by the attacker. The solid arrow denotes a successful connection and the dashed arrow denotes discovering new nodes.

Figure 8: **Example Interpretations**—Example interpretations for safe driving and lateral movement. The colored bar under each figure denotes the importance score of the time step derived by AIRS. The importance score is higher when the color goes from yellow to red.



(a) Original Episode



(b) Manipulated Episode

Figure 9: **Example Interpretation for One Episode in Lateral Movement**—The top figure shows the original episode and the step framed with a red rectangle is the critical time step identified by AIRS. The bottom figure shows the manipulated episode. We replay the episode until reaching the state with the blue background and replace the next action with a random action. Then the following actions are still computed by the policy network.

and $[4,3,2]$ can bring positive reward changes. To this end, we design the final solution which is to patch both states. This leads to a higher reward change (+0.301) as shown in the last row of the table. In this way, we can simply improve the agent's performance without retraining. Note that it is possible to leverage available domain knowledge to guide action selections and further improve the patching performance.

Compared to the re-training mechanism introduced above, patching is more lightweight. With limited computational resources, for tasks with small/discrete action spaces, patching is an alternative to re-training. However, for policies with large/continuous action spaces, it is hard to randomly search for proper patching action without guidance.

7.3 Application Debugging

By analyzing critical time steps, we can obtain a better understanding of the agent behavior. However, sometimes the identified time steps are not well aligned with human understanding or the application design. Using the case of network lateral movement, we illustrate how the explanation results can help to inform and correct application design mistakes.

Lateral Movement. During our analysis, we find the explanations of certain episodes are counter-intuitive. Figure 9(a) shows one of such examples. For this run, AIRS selects one intermediate time step as the most important time step. To confirm this state is indeed critical, we perform the fidelity evaluation as shown in Figure 9(b). The action in the selected critical state (with blue background) is replaced with a random action. After that, the DRL agent continues to take actions based on the policy network for the following states (yellow background). Interestingly, after replacing a single action for the selected state, in the new episode, the agent successfully took over the entire network and achieved the goal.

If we take a closer look at the original episode in Figure 9(a), we find that the agent keeps launching attacks against the already compromised nodes instead of further discovering new nodes, and thus fails to take control of the whole network. AIRS successfully identifies this "trap" state as the most important contributor to the final result.

We further analyze this "trap" state, and eventually correlate this behavior with two potential bugs in the reward design

Reward Design	Chain-Graph	Random-Graph
Original	89.3%	80.5%
Patch 1	99.3%	92.1%
Patch 2	99.6%	78.5%
Patch both	**99.7%**	**92.5%**

Table 4: **Evaluating Different Reward Designs**—For the lateral movement application, Patch 1 represents adding the time cost for any action. Patch 2 represents canceling the reward for attacks against already owned nodes. Patch both means the reward design combining both Patch 1 and Patch 2.

of the application. First, in the original reward design, we find that the time cost is not added to the total reward. Second, surprisingly, we find that launching attacks between two already owned nodes can bring in rewards. For these two reasons, the agent would repeatedly launch remote or local attacks even when the attack fails or the target node is already owned. We suspect these bugs were introduced due to developers' oversight during the reward function design.

We fix these bugs in the reward design and re-run an experiment to evaluate the correctness of our modifications. We train a DRL agent with the same parameters under the new reward designs for both the chain-graph and random-graph. We report the average attack success rate over 500 runs and the results are shown in Table 4. Although patching the second bug leads to a slight drop in the attack success rate in the random-graph[3], by patching both bugs, the attack success rate is much higher than the original reward design for both graphs. The result confirms the correctness of the patch.

After discovering these bugs in the reward design, we filed two bug reports to the development team (the numbers are omitted for anonymous submission). Microsoft have acknowledged our discoveries and labeled them as the enhancement to their simulator. This case study demonstrates how AIRS can help to debug and improve the application designs.

8 Discussion

Robustness of AIRS. To the best of our knowledge, [30] is the only existing attack against DRL explanation methods. It targets feature-level explanations and cannot be applied to our method. Like typical DNNs, AIRS can be vulnerable to data poisoning attacks. However, it is not clear how to define the proper threat model since the defender has full control over the environment and policy. Besides, as discussed in [59], the effectiveness of data poisoning attacks is influenced by many factors. Due to the ambiguities and uncertainties, we believe it requires non-trivial efforts to design proper data poisoning attacks against AIRS and leave it as our future work.

Connection to Robust DRL. As shown in Section 7.2, AIRS could explain and patch policy errors. Similarly, it can also be used to explain attacks and improve policy robustness. For example, existing work [77] develops an attack that trains an adversarial agent to disturb and fail a victim agent. AIRS can be used to explain why the victim agent fails and patch its weaknesses against the adversarial agent.

Transferability. AIRS is designed to train one explanation model for one policy. As such, The transferability of AIRS depends on the transferability of the policy. With a new environment, if the policy needed to be retrained, the explainer also needs to be retrained too. It does not jeopardize our practicability because the cost of training DNNs is much lower than training DRL policies.

Generalizability. As is specified in Section 4.1, our problem scope is DRL tasks that can be modeled as a single-agent Markov Decision Process (MDP). This could cover most existing DRL-based security applications. In addition, a few other security applications involve more than one agent (i.e., multi-agent autonomous driving systems [54, 76]). These applications are beyond our scope. As part of future work, we will investigate extending AIRS to these multi-agent environments. Note that we mainly focus on security applications in this work. For non-security applications with similar explanation requirements and RL models, our method may be applicable. But it cannot be applied to most non-security tasks with different environment models. For example, the GO game has a multi-agent extensive-form RL model, which is fundamentally different from our model. As such, AIRS cannot be adopted to explain DRL agents in this task (e.g., AlphaGo [61, 62]).

9 Conclusion

DRL has become a commonly adopted method for many security applications. To generate an explanation for DRL-driven security applications, we propose a new explainable method that establishes the relationship between the states and the final reward. We show that the explanation derived from our proposed approach can be potentially helpful for security professionals to understand the DRL model's behaviors better. Besides, we show that a high-fidelity explanation could be used to improve the effectiveness of DRL-driven security applications. With these observations, we conclude that explainability establishment could potentially become a driving force for the extensive adoption of DRL in security. As part of our future work, we intend to explore the utility of our proposed method from the user's perspective. Besides, we will also extend our method to other applications, exploring its effectiveness in (non-)security contexts.

[3] One possible explanation is that, without fixing the first bug (time penalty), only fixing the second bug (by canceling the reward for attacking already owned nodes) will make the reward much sparser. In other words, only few actions can receive the non-zero reward from the environment. In the chain-pattern graph, there are only two types of vulnerabilities. However, in the random-graph, there are more vulnerability types (i.e., much larger search space) which leads to degradation of agent performance.

References

[1] Iman Akbari, Ezzeldin Tahoun, Mohammad A Salahuddin, Noura Limam, and Raouf Boutaba. Atmos: Autonomous threat mitigation in sdn using reinforcement learning. In *Proc. of NOMS*, 2020.

[2] Jasem Almotiri, Khaled Elleithy, and Abdelrahman Elleithy. Comparison of autoencoder and principal component analysis followed by neural network for e-learning using handwritten recognition. In *Proc. of LISAT*, 2017.

[3] David Alvarez-Melis and Tommi S Jaakkola. Towards robust interpretability with self-explaining neural networks. In *Proc. of NeurIPS*, 2018.

[4] Dan Amir and Ofra Amir. Highlights: Summarizing agent behavior to people. In *Proc. of AAMAS*, 2018.

[5] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*, 2015.

[7] Tom Bewley and Jonathan Lawry. Tripletree: A versatile interpretable representation of black box agents and their environments. In *Proc. of AAAI*, 2021.

[8] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center*, 2016.

[9] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proc. of USENIX Security*, 2015.

[10] Ankur Chowdhary, Dijiang Huang, Jayasurya Sevalur Mahendran, Daniel Romo, Yuli Deng, and Abdulhakim Sabur. Autonomous security analysis and penetration testing. In *Proc. of MSN*, 2020.

[11] Youri Coppens, Kyriakos Efthymiadis, Tom Lenaerts, Ann Nowé, Tim Miller, Rosina Weber, and Daniele Magazzeni. Distilling deep reinforcement learning policies in soft decision trees. In *Proc. of IJCAI Workshop on XAI*, 2019.

[12] CrowdStrike. Lateral movement. url=https://www.crowdstrike.com/cybersecurity-101/lateral-movement/, 2019.

[13] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proc. of International Conference on Financial Cryptography and Data Security*, 2014.

[14] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proc. of ICCV*, 2017.

[15] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 1901.

[16] Andreas Fuchsberger. Intrusion detection systems and intrusion prevention systems. *Information Security Technical Report*, 2005.

[17] Bernhard C Geiger and Gernot Kubin. Relative information loss in the pca. In *Proc. of IEEE Information Theory Workshop*, 2012.

[18] Samuel Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding atari agents. In *Proc. of ICML*, 2018.

[19] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial examples for malware detection. In *Proc. of ESORICS*, 2017.

[20] Wenbo Guo, Sui Huang, Yunzhe Tao, Xinyu Xing, and Lin Lin. Explaining deep learning models-a bayesian non-parametric approach. *Proc. of NeurIPS*, 2018.

[21] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proc. of CCS*, 2018.

[22] Wenbo Guo, Xian Wu, Usmann Khan, and Xinyu Xing. Edge: Explaining deep reinforcement learning policies. *Proc. of NeurIPS*, 2021.

[23] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proc. of ICML*, 2018.

[24] Kim Hammar and Rolf Stadler. Finding effective security strategies through reinforcement learning and self-play. In *Proc. of CNSM*, 2020.

[25] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. Statistical learning with sparsity. *Monographs on statistics and applied probability*, 2015.

[26] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *Proc. of AAAI*, 2015.

[27] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proc. of CCS*, 2018.

[28] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proc. of AAAI*, 2018.

[29] Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. Squirrl: Automating attack analysis on blockchain incentive mechanisms with deep reinforcement learning. In *Proc. of NDSS*, 2019.

[30] Mengdi Huai, Jianhui Sun, Renqin Cai, Liuyi Yao, and Aidong Zhang. Malicious attacks against deep reinforcement learning interpretations. In *Proc. of KDD*, 2020.

[31] Sandy H Huang, Kush Bhatia, Pieter Abbeel, and Anca D Dragan. Establishing appropriate trust via critical states. In *Proc. of IROS*, 2018.

[32] Alihan Hüyük, Daniel Jarrett, Cem Tekin, and Mihaela Van Der Schaar. Explaining by imitating: Understanding decisions by interpretable policy learning. In *Proc. of ICLR*, 2021.

[33] Aya Abdelsalam Ismail, Hector Corrada Bravo, and Soheil Feizi. Improving deep learning interpretability by saliency guided training. In *Proc. of NeurIPS*, 2021.

[34] Aya Abdelsalam Ismail, Mohamed Gunady, Luiz Pessoa, Hector Corrada Bravo, and Soheil Feizi. Input-cell attention reduces vanishing saliency of recurrent neural networks. In *Proc. of NeurIPS*, 2019.

[35] Rahul Iyer, Yuezhang Li, Huao Li, Michael Lewis, Ramitha Sundar, and Katia Sycara. Transparency and explanation in deep reinforcement learning neural networks. In *Proc. of AIES*, 2018.

[36] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *Transactions on Intelligent Transportation Systems*, 2021.

[37] Quanyi Li, Zhenghao Peng, Zhenghai Xue, Qihang Zhang, and Bolei Zhou. Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. *arXiv preprint arXiv:2109.12674*, 2021.

[38] Quanyi Li, Zhenghao Peng, and Bolei Zhou. Efficient learning of safe driving policy via human-ai copilot optimization. *arXiv preprint arXiv:2202.10341*, 2022.

[39] Zenan Ling, Haotian Ma, Yu Yang, Robert C Qiu, Song-Chun Zhu, and Quanshi Zhang. Explaining alphago: Interpreting contextual effects in neural networks. *arXiv preprint arXiv:1901.02184*, 2019.

[40] Guiliang Liu, Oliver Schulte, Wang Zhu, and Qingcan Li. Toward interpretable deep reinforcement learning with linear model u-trees. In *Proc. of ECML-PKDD*, 2018.

[41] Guiliang Liu, Xiangyu Sun, Oliver Schulte, and Pascal Poupart. Learning tree interpretation from object representation for deep reinforcement learning. In *Proc. of NeurIPS*, 2021.

[42] Yang Young Lu, Wenbo Guo, Xinyu Xing, and William Stafford Noble. Dance: Enhancing saliency maps using decoys. In *Proc. of ICML*, 2021.

[43] Prashan Madumal, Tim Miller, Liz Sonenberg, and Frank Vetere. Distal explanations for model-free explainable reinforcement learning. *arXiv preprint arXiv:2001.10284*, 2020.

[44] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proc. of NDSS*, 2017.

[45] Microsoft. Cyberbattlesim: Gamifying machine learning for stronger security and ai models. url=https://github.com/microsoft/CyberBattleSim, 2021.

[46] 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 *Proc. of ICML*, 2016.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *Proc. of NeurIPS Deep Learning Workshop*, 2013.

[48] 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*, 2015.

[49] Alex Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo J Rezende. Towards interpretable reinforcement learning using attention augmented agents. In *Proc. of NeurIPS*, 2019.

[50] Alizée Pace, Alex J Chan, and Mihaela van der Schaar. Poetree: Interpretable policy learning with adaptive decision trees. In *Proc. of ICLR*, 2022.

[51] Zhenghao Peng, Quanyi Li, Ka Ming Hui, Chunxiao Liu, and Bolei Zhou. Learning to simulate self-driven particles system with coordinated policy optimization. In *Proc. of NeurIPS*, 2021.

[52] Zhenghao Peng, Quanyi Li, Chunxiao Liu, and Bolei Zhou. Safe driving via expert guided policy optimization. In *Proc. of Conference on Robot Learning*, 2022.

[53] Nikaash Puri, Sukriti Verma, Piyush Gupta, Dhruv Kayastha, Shripad Deshmukh, Balaji Krishnamurthy, and Sameer Singh. Explain your move: Understanding agent actions using specific and relevant feature attribution. In *Proc. of ICLR*, 2020.

[54] Anirban Santara, Sohan Rudra, Sree Aditya Buridi, Meha Kaushik, Abhishek Naik, Bharat Kaul, and Balaraman Ravindran. Madras: Multi agent driving simulator. *Journal of Artificial Intelligence Research*, 2021.

[55] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *Proc. of International Conference on Financial Cryptography and Data Security*, 2016.

[56] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proc. of ICML*, 2015.

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

[58] Jonathon Schwartz and Hanna Kurniawati. Autonomous penetration testing using reinforcement learning. *arXiv preprint arXiv:1905.05965*, 2019.

[59] Avi Schwarzschild, Micah Goldblum, Arjun Gupta, John P Dickerson, and Tom Goldstein. Just how toxic is data poisoning? a unified benchmark for backdoor and data poisoning attacks. In *Proc. of ICML*, 2021.

[60] Wenjie Shi, Shiji Song, Zhuoyuan Wang, and Gao Huang. Self-supervised discovering of causal features: Towards interpretable reinforcement learning. *arXiv preprint arXiv:2003.07069*, 2020.

[61] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 2016.

[62] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[63] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv:1312.6034*, 2013.

[64] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv:1706.03825*, 2017.

[65] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. of ICML*, 2017.

[66] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. In *Proc. of GECCO*, 2020.

[67] Nicholay Topin and Manuela Veloso. Generation of policy-level explanations for reinforcement learning. In *Proc. of AAAI*, 2019.

[68] TrendMicro. Lateral movement: How do threats actors move deeper into your network? url=http://about-threats.trendmicro.com/cloud-content/us/ent-primers/pdf/, 2013.

[69] Daniel Tschernutter, Tobias Hatt, and Stefan Feuerriegel. Interpretable off-policy learning via hyperbox search. In *Proc. of ICML*, 2022.

[70] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proc. of AAAI*, 2016.

[71] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proc. of ICML*, 2018.

[72] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 2019.

[73] Matt Vitelli and Aran Nayebi. Carma: A deep reinforcement learning approach to autonomous driving. *Tech. rep. Stanford University, Tech. Rep.*, 2016.

[74] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network

architectures for deep reinforcement learning. In *Proc. of ICML*, 2016.

[75] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. Evaluating explanation methods for deep learning in security. In *Proc. of EuroS&P*, 2020.

[76] Cathy Wu, Aboudy Kreidieh, Kanaad Parvate, Eugene Vinitsky, and Alexandre M Bayen. Flow: Architecture and benchmarking for reinforcement learning in traffic control. *arXiv preprint arXiv:1710.05465*, 2017.

[77] Xian Wu, Wenbo Guo, Hua Wei, and Xinyu Xing. Adversarial policy training against deep reinforcement learning. In *Proc. of USENIX Security Symposium*, 2021.

[78] Jiayun Xu, Yingjiu Li, and Robert H. Deng. Differential training: A generic framework to reduce label noises for android malware detection. In *Proc. of NDSS*, 2021.

[79] Ke Xu, Yingjiu Li, Robert H. Deng, Kai Chen, and Jiayun Xu. Droidevolver: Self-evolving android malware detection system. In *Proc. of EuroS&P*, 2019.

[80] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. Program synthesis guided reinforcement learning for partially observed environments. In *Proc. of NeurIPS*, 2021.

[81] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Deep reinforcement learning with relational inductive biases. In *Proc. of ICLR*, 2018.

[82] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware. In *Proc. of CCS*, 2020.

[83] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. Structural attack against graph based android malware detection. In *Proc. of CCS*, 2021.

A Additional Descriptions of Security Applications

Autonomous Driving. This application simulates a road environment where static and movable obstacles are randomly placed on the road. The self-driving agent's goal in this environment is to arrive at the destination safely. To guide the agent achieving this goal, the reward is designed as a summation of two components: (1) A reward that encourages the agent to drive forward and fast; (2) A reward that penalizes the agent when it violates traffic rules (i.e., Each time the agent violates a traffic rule, it receives a reward of -1, and If the vehicle can arrive at the destination without violation, it is rewarded by $+20$).

Malware Mutation. The agent's goal is to bypass the target malware detection model $f(\cdot)$ with minimal manipulations of malware graphs while maintaining the malware's functionality. According to [83], to maintain functionality, only four actions are allowed: "adding call function call", "rewiring function call", "inserting methods" and "deleting methods". To bypass $f(\cdot)$ with minimal manipulations, the reward function is designed as follows:

$$R = \begin{cases} 1 & if \ \ bypassing \ \ f(\cdot) \\ -(\Delta N_{edge} + \Delta N_{node}) & otherwise \end{cases} \quad (9)$$

where ΔN_{edge} and ΔN_{node} represent the number of edges and nodes manipulated. As we can easily tell from the above equation, to maximize its reward, the agent has to find the minimal manipulations required to bypass $f(\cdot)$.

Selfish Mining in Blockchain. In this application, the goal of a selfish miner is to maximize its rewarded blocks (i.e., the blocks mined by it and is connected to the longest chain). To do so, the agent could not only publish its mined blocks, but also hide them privately and publish them later. More specifically, the agent's action is designed as follows:

- **Adopting.** When the attacker chooses "adopting", the blocks in the private chain will be discarded, and the attacker accepts the public chain. This action is always feasible.
- **Waiting.** The "waiting" action is also always feasible that the attacker does nothing but keeps mining the block.
- **Overriding.** If $a > h$, the attacker can override the blocks in the public chain with his secret blocks.
- **Matching.** The attacker publishes conflicting blocks of the same length as that of the public chain. It is not always feasible. We will discuss the different conditions for "matching" below.

The state of this environment is the current status of the blockchain, and it is represented by a 3-element tuple $(a, h, fork)$. Here, a and h denote the length of the attacker's chain and the public chain after the latest fork. The element $fork \in \{relevant, irrelevant, active\}$ is a ternary value representing the status of the fork. For example, state $(a, h, relevant)$ means the newest block was mined by the honest miner. The previous state is $(a, h-1, \cdot)$ and if $a > h$, the attacker can choose to "match". Take another example, the state is $(a, h, irrelevant)$, the previous state is $(a-1, h, \cdot)$

	Selfish Mining	Chain-Graph	Radom-Graph	Safe Driving	Malware Mutation
Agent	DQN	DQN	DQN	PPO	DQN
$\theta(\cdot)$	FC(8,4,1)	FC(8,4,1)	FC(32,8,1)	FC(32,16,8,1)	FC(8,1)
$h(\cdot)$	FC(8,4,1)	FC(8,4,1)	FC(32,8,1)	FC(32,16,8,1)	FC(8,1)
Epoch	50	100	100	200	100

Table 5: Experimental setups: DRL algorithm, the neural network architecture, and the number of training epochs used for AIRS approximation model in each application. Note that "Chain-Graph" and "Random-Graph" represent two different topology setups for the lateral movement application [45].

Explainer	Random	Vanilla	Integrated	Smooth	Attention	LR	Highlights	Trust	AIRS
Reward Difference	1443.98	1395.43	1392.88	1345.50	1487.34	1394.87	1491.11	1421.42	**1539.05**
RRD	1	0.96637	0.96461	0.93179	1.03003	0.96598	1.03264	0.98438	**1.06584**

Table 6: Fidelity result for the random-graph network setting for lateral movement.

and the newest block was mined by the attacker. In this case, "matching" is infeasible since the honest minders have already received the h-th block. The *active* status of $fork$ means that the attacker has already chosen to do the "matching".

Network Lateral Movement. This application has two topology setups (i.e., environments): "Chain-Graph" and "Random-Graph". In the Chain-Graph network, the nodes are connected in a chain. Each node can only be accessed by the prior node and only has access to the next node. For each type of node, the vulnerabilities are also fixed (e.g., the vulnerability of the Windows node is "ScanExplorerRecentFiles" and the vulnerability of the Linux node is "ScanBashHistory"). The reward for connecting to the new node is also fixed as $+100$. When the attacker takes ownership of the whole network, it gains a great positive reward $+5000$ for the successful lateral movement. Instead, the Random-Graph network topology is generated randomly, and vulnerabilities of nodes are also defined at random. Nodes with the same type do not share the same vulnerabilities in the Random-Graph network. The values of each node are also randomly assigned, and thus, the rewards for connecting the new nodes are also random. They range from 10 to 100.

B Additional Experimental Setting

For the applications of selfish mining, network lateral movement, autonomous driving, and malware mutation, the default episode lengths are set as 15, 100, 1000, and 500, respectively, which are chosen according to the average length of the episodes. For a fair comparison, we apply the same state filter for all methods. We reserve a continuous sub-sequence with the highest correlation and filter out other states. The length of this sub-sequence is 50% of the episode length. We discussed how the length of this sub-sequence influences the explanation performance in Section 6.3. The DRL algorithm with the architectures and number of training epochs used for AIRS approximation model in each task are summarized in Table 5, where FC stands for the fully-connected layers. In all cases, we train the approximation model using the Adam optimizer with learning rate $l = 1e-3$ and the coefficient $\lambda = 1e-6$ for the regularization term in Section 4.4. All of our experiments are run on a server with 2 AMD EPYC 7702 64-Core CPU Processors and 4 NVIDIA RTX A6000 GPUs. For reproducibility, we have released our codes at https://github.com/sherdencooper/AIRS.

C Exhaustive Search

The exhaustive search seems to be a simple method to identify the critical steps in the episode. However, when doing the exhaustive search, the episode needs to be replayed many times to identify the important step. When the episode length is long, the time cost is not acceptable to derive nearly real-time explanations. Moreover, when the interaction with the environment is time-consuming, the exhaustive search is not feasible because it needs a lot of interactions with the environment to generate the explanations when the DRL agent is running. We compare the efficiency between the exhaustive search and AIRS in Table 7. We can find that even in the selfish mining application, the time cost is unacceptable. In complex applications, it takes days to generate the explanation. Also, if the explainer is asked to identify multiple critical sub-sequences in the episode, the time cost for exhaustive search is even larger since it cannot capture the dependency between sub-sequences. Therefore, we typically do not use the exhaustive search in the deployment.

Explainer	Selfish(s)	Chain(s)	Random(s)	Safe(s)	Malware(s)
AIRS	143.94	194.07	448.35	790.63	225.20
Exhaustive Search	1074.72	7690.46	13524.75	512175.64	97106.32

Table 7: Time cost evaluation for generating explanations for 500 episodes between AIRS and the exhaustive search. The time cost for AIRS contains both training time cost and generation time cost.