Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Mind the Gap: towards generalizable autonomous penetration testing via domain randomization and meta-reinforcement learning

Shicheng ZHOU[†1,2], Jingju LIU[†‡1,2,3], Yuliang LU[†‡1,2], Jiahai YANG[3], Yue ZHANG[†‡4], Jie CHEN[1]

[1]*College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China*

[2]*Anhui Province Key Laboratory of Cyberspace Security Situation, Awareness and Evaluation, Hefei 230037, China*

[3]*Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China*

[4]*College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*

[†]E-mail: zhoushicheng@nudt.edu.cn; liujingju17@nudt.edu.cn; luyuliang@nudt.edu.cn; zhangyue@nudt.edu.cn

Received Feb. 16, 2025; Revision accepted Oct. 17, 2025; Crosschecked Nov. 27, 2025; Published online Dec. 12, 2025

Abstract: With the increasing number of vulnerabilities exposed on the Internet, autonomous penetration testing (pentesting) has emerged as a promising research area. Reinforcement learning (RL) is a natural fit for studying this topic. However, two key challenges limit the applicability of RL-based autonomous pentesting in real-world scenarios: the training environment dilemma—training agents in simulated environments is sample-efficient while ensuring that their realism remains challenging; poor generalization ability—agents' policies often perform poorly when transferred to unseen scenarios, with even slight changes potentially causing a significant generalization gap. To address both challenges, we propose a generalizable autonomous pentesting framework termed GAP, which aims to achieve efficient policy training in realistic environments and train generalizable agents capable of drawing inferences about other cases from one instance. GAP introduces a real-to-sim-to-real pipeline that enables end-to-end policy learning in unknown real environments while constructing realistic simulations and improves agents' generalization ability by leveraging domain randomization and meta-RL learning. We are among the first to apply domain randomization in autonomous pentesting and propose a large language model-powered domain randomization method for synthetic environment generation. We further apply meta-RL to improve agents' generalization ability in unseen environments by leveraging synthetic environments. Combining the two methods effectively bridges the generalization gap and improves agents' policy adaptation performance. Simulations are conducted on various vulnerable virtual machines, with results showing that GAP can enable policy learning in various realistic environments, achieve zero-shot policy transfer in similar environments, and achieve rapid policy adaptation in dissimilar environments.

Key words: Cybersecurity; Penetration testing; Reinforcement learning; Domain randomization; Meta-reinforcement learning; Large language model

https://doi.org/10.1631/FITEE.2500100 **CLC number:** TP393

1 Introduction

Penetration testing, shortly "pentesting" or PT, is an effective methodology to assess cybersecurity through authorized simulated cyberattacks. It aims to preemptively identify security vulnerabilities, allowing organizations to proactively enhance their security measures and defenses. However, with

‡ Corresponding authors

ORCID: Shicheng ZHOU, https://orcid.org/0000-0001-9686-3836; Jingju LIU, https://orcid.org/0009-0005-9506-6903; Yuliang LU, https://orcid.org/0000-0002-8502-9907; Yue ZHANG, https://orcid.org/0009-0007-3570-2132

© Zhejiang University Press 2025

more and more vulnerabilities exposed on the Internet, traditional manual-based pentesting has become more costly, time-consuming, and personnel-constrained (Holm, 2023). Considering this, autonomous pentesting has emerged as a promising research area. Pentesting can be seen as a dynamic sequential decision-making process, while reinforcement learning (RL) is a suitable method for optimizing such decisions. RL trains an agent to learn a policy through trial and error by interacting with environments, without the need for supervision or predefined environmental models. This makes RL a natural fit for studying autonomous pentesting.

RL-based autonomous pentesting aims to train agents to learn how to explore and exploit vulnerabilities in target hosts. However, achieving satisfactory performance with RL algorithms often requires a large number of training samples (Chen XY et al., 2022). This is typically impractical in autonomous pentesting, where agents' interactions with real-world environments are time-consuming and risky due to action execution and network latency. A common way to tackle this issue is to train the agents in simulated or emulated environments and subsequently transfer the learned policy to real-world scenarios. But this way comes with two new challenges.

Challenge 1: training environment dilemma. This dilemma represents a conflict between the realism of the training environment and the efficiency of the training process. In previous research on RL-based autonomous pentesting, the training environments for agents have typically taken two forms. One approach (Tran et al., 2021) is to build simulated training environments based on network simulators e.g., NetworkAttactSimulator (NASim) (Jonathon and Hanna, 2019), wherein agents interact with the environment logically. Training agents in such simulated environments is sample-efficient, yet ensuring that their realism remains challenging. By contrast, the other approach (Takaesu, 2018) involves training agents directly in emulated environments (e.g., virtual machines), which closely mimics real-world settings but suffers from lower training efficiency and limited diversity. To tackle this challenge, one can improve the agents' learning efficiency in emulated environments or construct simulated environments close to real-world settings.

Challenge 2: poor generalization ability. That is, even with suitable training environments, agents' learned policies often exhibit poor performance when transferred to unseen testing (real-world) scenarios (Wang et al., 2020). The worst of it is that even a slight change in the environment would require the agents' policy to be retrained. This phenomenon is known as the generalization gap (Kirk et al., 2023), which significantly constrains the widespread applicability of pentesting agents. To tackle this, one needs to improve agents' generalization ability. The reasons for the generalization gap are twofold:

1. RL algorithms tend to overfit the training environments (Cobbe et al., 2019);

2. Training environments have limited diversity, whereas real-world environments are unknown for agents and unpredictably diverse. The differences between them are known as the reality gap (Tobin et al., 2017).

Fig. 1 presents an example of the reality gap and generalization gap encountered in the pentesting field. In Fig. 1, the training environment is a virtual machine with the CVE-2018-7600 vulnerability, set up using Vulhub (https://github.com/vulhub/vulhub). We use the Shodan (https://www.shodan.io) engine to search real-world hosts that may possess the same vulnerability, which is assumed to be a testing environment. From Fig. 1a, we can see that even if vulnerabilities are the same between training and real-world testing environments, the configurations of hosts can differ. For instance, vulnerable products (VPs) may be exposed on various ports, hosts could be running different operating systems (OSs), and there might also be numerous other ports open that are unrelated to the vulnerabilities. These differences create a gap between these environments, leading to the agent perceiving different observations. Then, a generalization gap will occur once the agent learns to rely on certain observational features in the training environment that change in the testing environment. Fig. 1b visually demonstrates the impact of the generalization gap on the agent's policy transfer. We train the agent using APRIL (Zhou et al., 2024) in the training environment, and then transfer the trained agent to the testing environment for evaluation. The zero-shot performance in the testing environment shows a significant decline. We further demonstrate this phenomenon in Section 6.2.

We argue that an agent with good generalization

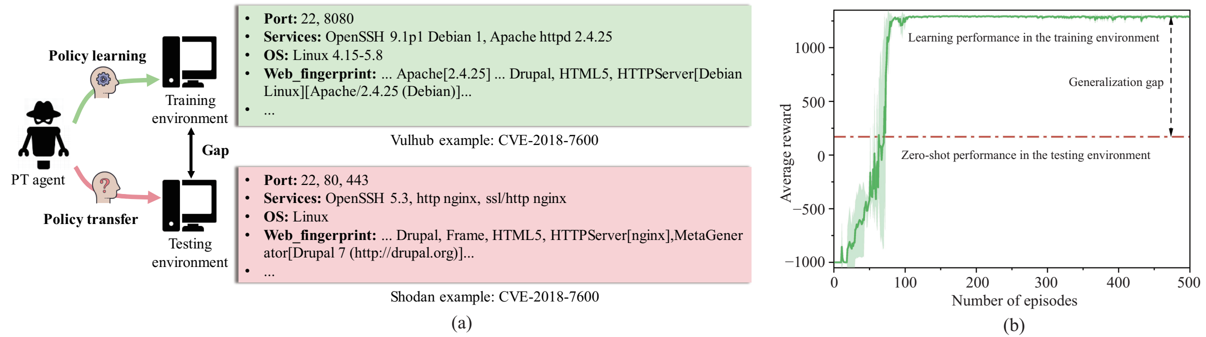

Fig. 1 Gaps between the training and testing environments: (a) reality gap where hosts with the same vulnerability have different configurations; (b) generalization gap that shows the learning curve of APRIL in the training environment and its zero-shot performance in the testing environment

ability should be able to draw inferences about other cases from one instance, akin to human capabilities. Specifically, when the testing environments are similar to the training environment, that is, both have the same vulnerabilities, the agent should be able to bridge the generalization gap and achieve zero-shot policy transfer. Moreover, it should demonstrate the capacity for few-shot policy adaptation in dissimilar scenarios featuring varying vulnerabilities, thereby enhancing the overall learning efficiency.

To tackle these two challenges, we propose a generalizable autonomous pentesting framework, namely GAP. GAP works following a real-to-sim-to-real pipeline, wherein the training environment dilemma and poor generalization ability can be tackled. In the real-to-sim phase, GAP achieves end-to-end learning in unknown real/emulated environments without requiring significant human effort, while constructing realistic simulation analogs for real environments. On this basis, in the sim-to-real phase, GAP aims to enhance agents' generalization ability, enabling them to achieve two key objectives: (1) bridging the generalization gap for zero-shot policy transfer in similar scenarios and (2) facilitating rapid policy adaptation in dissimilar scenarios to enhance learning efficiency. For this purpose, the following two methods are applied:

1. Environment augmentation. Similar to data augmentation in supervised or unsupervised learning, domain randomization (Chen XY et al., 2022; Horváth et al., 2023) can be considered a form of environment augmentation technique that is widely used in robotics to improve the robustness and generalization of models by highly randomizing the rendering settings for the simulated training set. Moreover, large language models (LLMs) have recently demonstrated an unprecedented ability to generate synthetic data for specific tasks (Li ZY et al., 2023; Guo and Chen, 2024), providing us with a promising approach to domain randomization. Inspired by these, we propose an LLM-powered domain randomization method to synthesize sufficient simulated training environments based on limited real-world data. This approach offers a feasible solution to increase both the number and the diversity of training environments. By using domain randomization to randomly change part of the host configuration during training, the agent can prevent its policy from overfitting to specific observational features, thereby enhancing policy robustness.

2. Learning to learn. Meta-learning (Ye et al., 2024), also known as learning to learn, aims to enable models to adapt to unseen tasks quickly by leveraging prior learning experiences on multiple training tasks. Meta-RL aims to apply this principle to RL. Recent advancements highlight its potential to address RL's inherent overfitting and sample inefficiency (Finn et al., 2017; Beck et al., 2023). While meta-RL facilitates efficient adaptation of agents to unseen testing environments, it necessitates extensive meta-training across diverse related tasks. However, due to the absence of appropriate training environments, meta-RL has not been successfully applied in autonomous pentesting before. To address this challenge, this paper initially uses domain randomization to create an adequate number of simulated training environments. Subsequently, it applies meta-RL to extract inductive biases from these synthetic environments, thereby enhancing the agents' generalization ability.

To sum up, our main contributions are threefold:

1. We propose GAP, a generalizable autonomous pentesting framework that works on a real-to-sim-to-real pipeline. This framework enables end-to-end policy learning in unknown real environments as well as the construction of realistic simulations, while improving agents' generalization ability.

2. To bridge the generalization gap and achieve fast policy adaptation, we are among the first to apply domain randomization in the autonomous pentesting domain and propose an LLM-powered domain randomization method for environment augmentation. We further apply meta-RL to improve the agents' generalization ability to unseen environments by leveraging the generated simulations.

3. We conduct simulations on various vulnerable virtual machines, with results showing that GAP can enable policy learning in various realistic environments, achieve zero-shot policy transfer in similar environments, and achieve rapid policy adaptation in dissimilar environments.

2 Related works

Recently, RL has shown remarkable progress in various domains, including autonomous vehicles (Feng et al., 2023), unmanned aerial vehicles (Bo et al., 2024), and cybersecurity (Chen J et al., 2023). Compared to traditional planning methods, RL allows for training an agent to learn a policy through trial and error while interacting with the environment, eliminating the need for supervision or environmental models. This agent–environment interaction learning paradigm makes RL suitable for studying autonomous pentesting.

Part of previous research on RL-based autonomous pentesting mainly focused on training agents to discover attack paths in pre-configured static simulations based on network simulators, e.g., NASim (Jonathon and Hanna, 2019) and CyberBattleSim (Microsoft Defender Research Team, 2021). In terms of training environment, these simulations often abstract specific scenarios using simplified models, such as coarse-grained representations of host information and actions, e.g., basic operating system (OS) types and generalized vulnerability exploits. As a result, these environments are generally limited to specific white-box pentesting scenarios, where researchers are assumed to have prior knowledge of the target environment's configuration and even vulnerabilities. This over-optimistic and unrealistic assumption, while useful in controlled settings, does not hold for real-world black-box pentesting, where the environment's details are unknown (Zhou et al., 2024). Furthermore, the abstraction in agents' state and action spaces of these simulators leads to another limitation: agents trained in such environments often struggle to apply their learned policies in real environments. For example, coarse actions ("exploit HTTP" action in NASim) or generalized host information fails to capture the nuance of real environments, where agents must react to fine-grained conditions. While recent works, such as that on PenGym (Nguyen et al., 2025), have made strides in incorporating real-world tools and feedback through the use of virtualization, e.g., kernel-based virtual machine (KVM), they are still constrained by the need for pre-defined, specific simulation scenarios (e.g., NASim). They also continue to rely on prior knowledge of the environment. In terms of training algorithms, researchers have made attempts to improve the learning efficiency of the pentesting agents through various methods, such as action space decomposition (Tran et al., 2021), distributed algorithms (Ilic et al., 2024), or the introduction of demonstration data (Chen JY et al., 2023) and curiosity mechanisms (Yang et al., 2025). Besides, Yang et al. (2023) and Li QY et al. (2024) have provided solutions to enhance the adaptability of agents in response to dynamic changes in such simulated environments. Remarkable progress has been made, though, they failed to adequately address the issue of agents' poor generalization ability.

There has also been some research (Takaesu, 2018; Maeda and Mimura, 2021; Zhou et al., 2024) attempted to train pentesting agents in emulated environments (like virtual machines) that closely mimic real-world settings. However, these efforts often trained agents to learn policies in specific target machines. Such learned policies frequently suffer from poor generalization, as they tend to overfit specific training scenarios or tasks. Consequently, even a slight change in the environment would require agents' policies to be retrained. This limitation hinders their adaptability to the diverse vulnerability environments encountered in the real world, posing challenges to the widespread deployment of pentesting agents.

Compared to previous research, this paper focuses on enabling agents to train in emulated environments (real vulnerable virtual machines) that are close to real-world settings. We construct simulated environments by collecting data from agents' interactions with those real vulnerable environments. The constructed simulated environments incorporate feedback from the real environments, allowing agents to receive the same feedback when interacting with the simulated environment as they would in the real environment. This ensures that training in the real environment and training in the simulated environment yield equivalent results. More importantly, this approach helps avoid high time costs associated with repeated interactions, enhancing the learning efficiency of agent training. Additionally, it serves as a digital replica of the real environment, providing environmental samples for subsequent research. The proposed method does not require prior knowledge of the target environment; instead, it dynamically constructs the simulated environment based on the agent's actual interactions with the real environment. Its goal is to provide efficient and realistic environmental support for subsequent repetitive training or environment augmentation.

Moreover, this work follows the settings in Zhou et al. (2024), with another goal of training pentesting agents that can be generalized to unseen scenarios. This generalization ability refers to drawing inferences from one instance to another, enabling agents to bridge the generalization gap for zero-shot policy transfer in similar environments with the same vulnerabilities, as well as achieve few-shot policy adaptation in dissimilar scenarios.

3 Preliminary

3.1 RL approach

RL is a machine learning method that maps the state of the environment to actions. Markov decision processes (MDPs) serve as the mathematical foundation for RL algorithms, providing a formal framework for modeling sequential decision-making.

A discrete-time MDP can be formalized by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, where $\mathcal{S}$ represents the state space, $\mathcal{A}$ is the action space, $\mathcal{R}: \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is the reward function, $\mathcal{P}: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0,1]$ is the state transition probability function, and γ is the discount factor used to determine the importance of long-term rewards. For a standard RL problem, at each time step t, the RL agent observes the state s_t from the environment and selects an action a_t based on its policy π. Then, the agent receives a reward signal from the environment, and the environment transitions to the next state s_{t+1}. The goal of the RL agent is to learn the optimal policy π^* with parameters ϕ^* that maximizes the expected discounted reward within an episode. Thus, the RL objective function is defined as follows:

$$J(\pi_\phi) = \mathbb{E}_{\tau \sim P_{\pi_\phi}} \left[\sum_{t=0}^{T} \gamma^t r_t \right], \tag{1}$$

where $\gamma \in [0,1]$ is the discount factor used to determine the importance of long-term rewards, T is the horizon, $\tau = \{s_t, a_t, r_t\}_{t=0}^{T}$ is the T-step trajectory, and P_{π_ϕ} denotes the distribution of τ under π_ϕ.

3.2 Meta-RL approach

Meta-learning, often framed as learning to learn, aims to learn a general-purpose learning algorithm by leveraging a set of tasks with a shared structure in the training phase, so that it can generalize well to new and similar tasks (Hospedales et al., 2022). Meta-RL, in short, involves applying this concept to RL.

Generally, considering the parameterized RL policies with parameters $\phi \in \varPhi$, we can define an RL algorithm as the function f that maps the training data to a policy: $f(D) = ((\mathcal{S} \times \mathcal{A} \times \mathbb{R})^T)^H \rightarrow \varPhi$, where H denotes the total number of training episodes and $D = \left\{\tau^h\right\}_{h=0}^{H}$ represents all of the trajectories collected in the MDP. The main idea of meta-RL is instead to learn the RL algorithm f that outputs the parameters of the RL policy (Beck et al., 2023): $\phi = f_\theta(\mathcal{D})$, where θ represents the meta-parameters which need to learn to maximize a meta-RL objective, and $\mathcal{D}$ represents the meta-trajectories that may contain multiple policy trajectories.

Meta-RL usually involves two loops of learning: an inner loop, where the agent adapts its policy to a specific task, and an outer loop, also referring to meta-training, where the agent updates its meta-parameters θ based on the performance across multiple tasks to allow the agent to quickly adapt to or learn efficiently in new tasks. Meta-training requires access to a set of training tasks/environments, each of which is formalized as an MDP and comes

from a distribution denoted as $p(\mathcal{M})$. In this paper, the distribution is defined over different variations (e.g., different host configurations) of the pentesting tasks, which differ only in state transition probability while maintaining consistency in $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$. Thus, meta-training can be described as a process that proceeds by sampling a task from $p(\mathcal{M})$, executing the inner loop on it, and optimizing the inner loop to improve policy adaptation. Formally, following the work (Beck et al., 2023), we define the meta-RL objective as follows:

$$\mathcal{J}(\theta) = \mathbb{E}_{\mathcal{M}_i \sim p(\mathcal{M})} \left[\mathbb{E}_{\mathcal{D}} \left[\sum_{\tau \in \mathcal{D}} G(\tau) \middle| f_\theta, \mathcal{M}_i \right] \right], \quad (2)$$

where $G(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ is the discounted reward in task $\mathcal{M}_i$, and τ is the trajectory sampled from task $\mathcal{M}_i$ under the policy with parameters $\phi = f_\theta(\mathcal{D})$.

3.3 Generalization in RL

To evaluate the generalization ability of pentesting agents, we train the agent on a set of environments $\mathcal{M}_{\text{train}} = \{\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_n\}$ with $\mathcal{M}_i \sim p(\mathcal{M})$, and then we evaluate its generalization performance on the testing environments $\mathcal{M}_{\text{test}}$ drawn from $p(\mathcal{M})$.

We evaluate the agents' generalization ability in two ways. First, following the work (Kirk et al., 2023), we study the zero-shot policy transfer performance in testing environments using the generalization gap, which is defined as

$$\begin{aligned} &\text{GenGap}(\pi) \quad (3) \\ =&\mathbb{E}_{\mathcal{M}_{\text{train}}, \tau \sim P_\pi} \left[G(\tau) \right] - \mathbb{E}_{\mathcal{M}_{\text{test}}, \tau \sim P_\pi} \left[G(\tau) \right], \end{aligned}$$

where $G(\tau)$ is the expected cumulative reward over a trajectory, and $\mathbb{E}_{\mathcal{M}_{\text{test}}, \tau \sim P_\pi} [G(\tau)]$ is the zero-shot performance. This metric measures the agent's ability to overcome overfitting and achieve zero-shot generalization. A smaller generalization gap indicates that the deployment performance will not deviate as much from the training performance, potentially indicating a more robust policy. In addition, following the work (Kirk et al., 2023), we study the few-shot policy adaptation performance by allowing the agent to interact with the testing environment online. This metric can evaluate how well the agent can quickly adapt its learned policy to new tasks or environments with stronger forms of variation.

4 Proposed method

4.1 Overview

GAP aims to achieve efficient policy training in realistic environments and train generalizable agents capable of drawing inferences about other cases from one instance. "Learning from one instance" is the foundational skill, while "generalizing to another" is the core objective. To achieve this goal, it employs a real-to-sim-to-real pipeline that integrates end-to-end policy learning, the construction of realistic simulation environments, and the enhancement of the agent's generalization ability. The full workflow of GAP is illustrated in Fig. 2.

4.1.1 Real-to-sim

The increasing emergence of vulnerabilities, coupled with the complex and dynamic nature of real-world network environments, necessitates the ability of pentesting agents to learn and adapt to policies in an end-to-end manner within unknown settings. GAP achieves this through the real-to-sim stage (see Section 4.2 for details), as shown by the blue lines in Fig. 2: ① end-to-end policy learning in the original training environment and ② realistic simulation construction based on the information gathered during policy learning. This constructed simulation serves as a digital mapping of the real environment, facilitating efficient policy training and environment augmentation in the subsequent stage.

4.1.2 Sim-to-real

Sim-to-real is a comprehensive concept that has been applied to robotic and classic machine vision tasks (Zhao et al., 2020). The goal of sim-to-real transfer is to ensure that the behaviors, actions, or decisions learned in simulations can effectively and reliably be applied in real-world scenarios. Similarly, in the sim-to-real stage, GAP uses the simulated environment constructed in the last stage to improve the agent's generalization ability, allowing the agent to quickly adapt to diverse real-world settings. As shown by the red lines in Fig. 2, GAP achieves this through two-phase steps: ③ synthetic environment generation via LLM-powered domain randomization (see Section 4.3) and ④ meta-RL training based on the synthetic environments (see Section 4.4). In this way, the agent is expected to achieve zero-shot policy

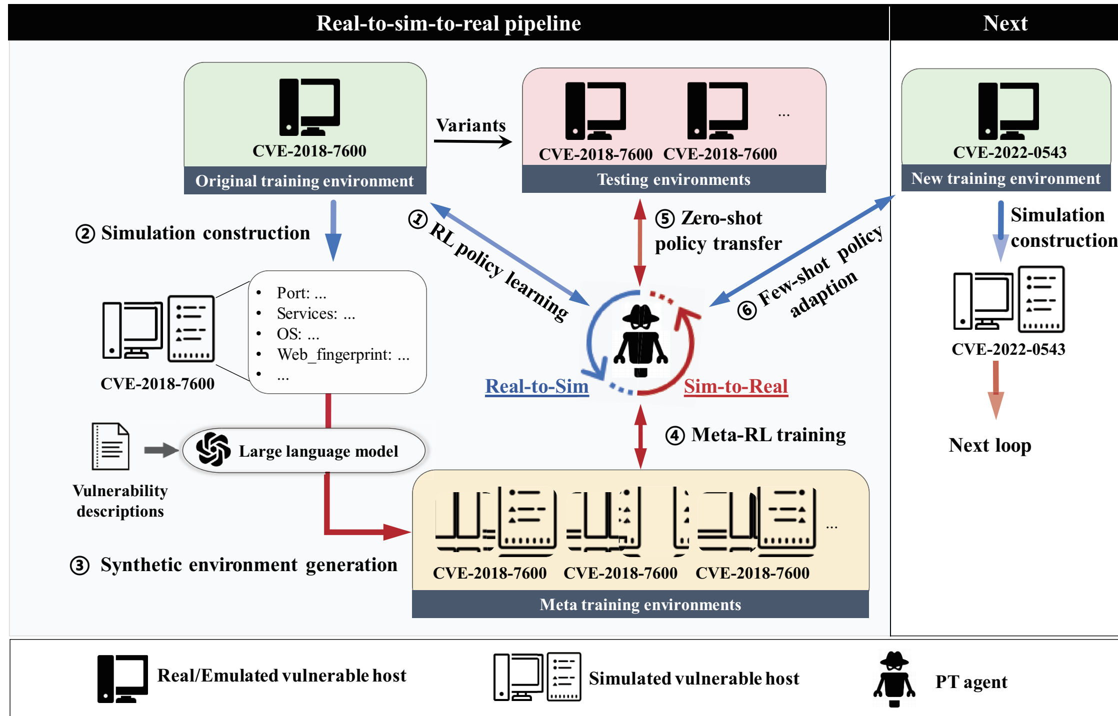

Fig. 2 Overview of GAP. References to color refer to the online version of this figure

transfer in similar environments (step ⑤ in Fig. 2) and few-shot policy adaptation in dissimilar environments (step ⑥ in Fig. 2).

Note that in our setup, we use virtual machines as surrogate real-world environments to train and test pentesting agents, since they provide controlled, isolated, and high-fidelity environments for agents to practice and refine offensive security techniques. The testing environments are variants of the original training environment, featuring the same vulnerability but different host configurations.

To make a clear distinction between different environments in this paper, we use the symbol $\mathcal{M}_h^k$ to denote a host h with vulnerability k. Thus, the original training environment is denoted as $\mathcal{M}_0^k$, the simulation of $\mathcal{M}_0^k$ is denoted as $\widetilde{\mathcal{M}}_0^k$, and the set of generated synthetic environments is denoted as $\widetilde{\mathcal{M}}^k = \{\widetilde{\mathcal{M}}_1^k, \widetilde{\mathcal{M}}_2^k, ...\}$. To simplify the symbolic representation, we omit the vulnerability identifier k in subsequent sections.

4.2 Policy learning and construction of simulated environments

Being able to learn policy in unknown environments is a fundamental skill for pentesting agents, where unknown environments refer to scenarios where the agent lacks prior knowledge of the target hosts' configuration and vulnerabilities. In this part, we introduce how we model the pentesting process as an MDP, thereby employing RL to train the agent to learn how to explore and exploit vulnerabilities using observed environmental states in an end-to-end manner. Additionally, upon detecting host configuration details, the agent constructs simulated environments to enhance its policy generalization capabilities in subsequent stages. The process of policy learning and construction of simulated environments is depicted in Fig. 3.

4.2.1 Model pentesting as an MDP

To train the pentesting agent using RL algorithms, we model the pentesting process as an MDP by following the idea of Zhou et al. (2024). This is formalized by defining the state space $\mathcal{S}$, action space $\mathcal{A}$, and reward function $\mathcal{R}$. For model-free RL algorithms, the transition probability function $\mathcal{P}$ remains unknown.

$\mathcal{S}$ comprises all potential environmental states observable by the agent. These observations typically involve host configurations detectable using

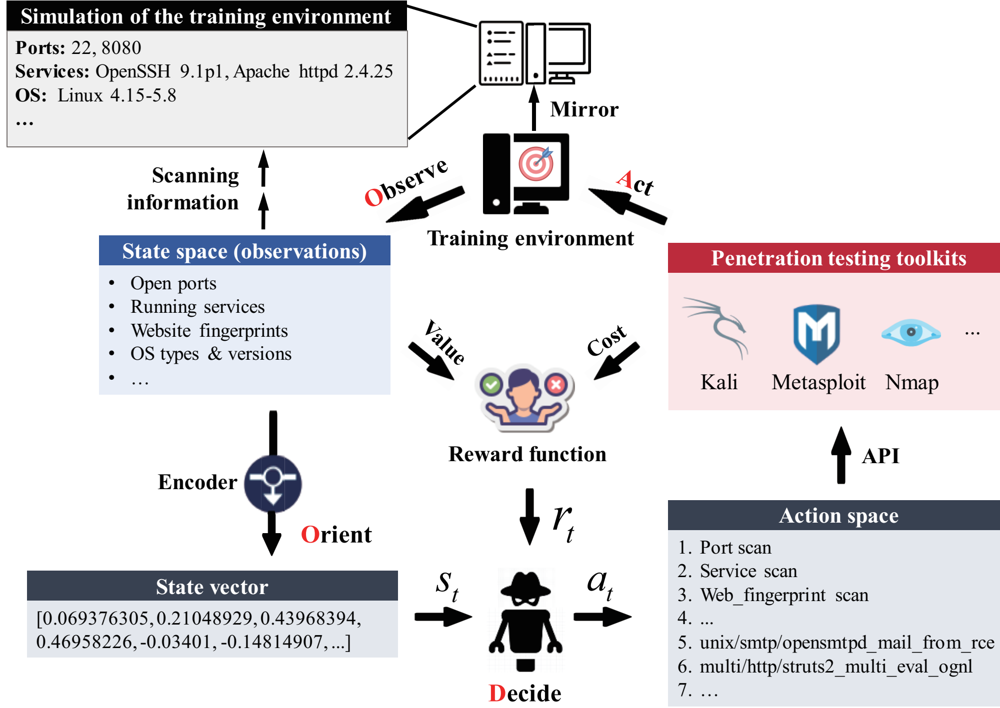

Fig. 3 The process of policy learning and construction of simulated environments

scanning tools, including scanning information such as ports, services, OSs, and website fingerprints. Human experts can use this information to pinpoint potential vulnerabilities within the target host, and thus, these details can form the foundation for the agent to make decisions.

$\mathcal{A}$ represents the set of all actions available to the agent. For pentesting tasks, these actions usually include various forms of information gathering, system probing, and vulnerability exploitation. In contrast to previous research that used abstract actions, our approach emphasizes the use of concrete and executable actions for end-to-end autonomous pentesting. These actions are aligned with specific commands or payloads derived from real pentesting tools or systems, such as Kali, Metasploit, and Nmap. Note that a larger action space necessitates more extensive exploration for agents to discover effective actions, thereby potentially reducing the tractability of the agent's training process. This phenomenon can be seen in simulations in Section 6.1.

$\mathcal{R}$ defines the agent's learning goal. In our study, the agent aims to gather critical information from the target host to accurately pinpoint the potential vulnerability and exploit it to compromise the host. Throughout this process, the agent should minimize the overall cost of actions to maintain its covert nature. For this purpose, we define the reward function as

$$\mathcal{R} = \text{value}(h) - \textstyle\sum_{a\in A}\text{cost}(a), \tag{4}$$

where $A \subseteq \mathcal{A}$ is the set of actions used in the pentesting process, value(h) returns a positive reward value if the target host h is compromised by exploiting the correct vulnerability or the agent successfully gains some kind of information about the host, and cost(a) refers to the cost of action a.

At its core, the process of interaction between the agent and its environment can be likened, to some extent, to an observe, orient, decide, and act (OODA) loop. This process can be described as follows:

First, the agent uses scanning tools to observe environmental states and collect raw observations, typically in textual format. This data often contains significant redundancy and cannot be directly fed into the agent's policy model. Therefore, the agent needs to orient the raw observations by analyzing and synthesizing the observed information to construct a state vector that the neural network can comprehend. To achieve this, we use the pre-trained Sentence-BERT (Wang et al., 2021) model as the

encoder for embedding the raw observations. These embedded representations of state vectors serve several purposes: (1) capturing essential features of the raw observations in a latent space, (2) facilitating end-to-end learning, and (3) preventing the dimensional explosion of the state space. Thus, the agent takes the state vector as input, makes a decision, and then outputs an action from the action space. Based on the output action, the agent acts on the target host by invoking and executing the corresponding tools in pentesting toolkits via the application programming interface (API). Finally, the interaction process transitions into the next loop.

4.2.2 Construction of simulated environments

During interaction with the original training environment $\mathcal{M}_0$, the agent uses gathered raw observations to build a simulated environment $\widetilde{\mathcal{M}}_0$ in JavaScript object notation (JSON) format. These observations contain host configuration data such as open ports, running services, and website fingerprints. The constructed simulated environments incorporate feedback from the original training environment, allowing agents to receive the same feedback when interacting with the simulated environment as they would in the original training environment. Thus, theoretically, the agent interacting with this simulated environment equates to interacting with a real environment. More importantly, this simulated environment can be used not only for efficient training and validation of the agent's policies but also as a real-world environment example for subsequent environment augmentation. Fig. 3 shows the process of constructing the simulated environment, while all constructed simulated environments can be found in our code repository.

4.2.3 Policy training

We train the agent's policy using the proximal policy optimization (PPO) algorithm (Schulman et al., 2017) on the original training environment $\mathcal{M}_0$. It is noteworthy that GAP is not limited to PPO. A wide range of policy gradient-based RL algorithms can be used out of the box.

PPO is an on-policy actor–critic method with two primary variants of PPO: PPO-Penalty and PPO-Clip; here, we focus on the most widely used variant PPO-Clip. By using PPO, the agent updates policy π_ϕ by taking multiple steps of (usually minibatch) stochastic gradient descent (SGD) to optimize the following objective:

$$\begin{aligned}&\mathcal{L}^{\mathrm{PPO}}(\phi)\\&=\mathbb{E}_{\tau_t\sim\pi}\left[\min\left(r_t(\phi)\hat{A}_t,\mathrm{clip}(r_t(\phi),1-\epsilon,1+\epsilon)\hat{A}_t\right)\right],\end{aligned}\tag{5}$$

where $r_t(\phi)=\frac{\pi_\phi(a_t|s_t)}{\pi_{\phi_{\mathrm{old}}}(a_t|s_t)}$ denotes the probability ratio, $\hat{A}_t$ is an estimator of the advantage function which describes how much better the action is than others on average, $\mathrm{clip}\left(r_t(\phi),1-\epsilon,1+\epsilon\right)$ modifies the surrogate objective by clipping the probability ratio, and ϵ is a small hyperparameter which roughly measures how far away the new policy is allowed to go from the old.

4.3 LLM-powered domain randomization

In the real world, hosts with the same vulnerability are affected by the same VP, but their configurations can differ significantly (see Fig. 1). They may run different versions of VP or services, have varied open ports, operate on different OSs, or exhibit different website fingerprints. These differences create the reality gap, meaning that an agent trained in a single environment would rely on specific observations and thereby may struggle to directly transfer its learned policies to real-world deployment. Therefore, considering and simulating these real-world differences during training is crucial to enhancing the generalization ability of pentesting agents.

In this paper, we use domain randomization as an environment augmentation technique, which can be feasible to address the challenge posed by limited diversity in training environments. Unlike vision-based tasks, wherein domain randomization is commonly applied, pentesting agents gather information through scanning tools that provide textual feedback. Textual data encompasses semantic, syntactic, and logical structures of language rather than straightforward physical attributes. This abstract nature complicates the randomization of text data, as any variation must preserve semantic coherence, specificity, and comprehensibility. The recent advancements in LLMs demonstrate the capability to generate synthetic data that mimics the characteristics and patterns of real-world data, thereby being a suitable and promising approach to performing domain randomization for autonomous pentesting.

Vulnerability descriptions in the National Vulnerability Database (NVD) repository detail key characteristics such as VPs, affected versions, product vendors, and impact. The constructed simulation of the original training environment serves as a realistic example. Therefore, we use these descriptions as background knowledge and the constructed simulation as task examples to design prompts for LLMs. The prompt pattern is shown in Fig. 4.

Fig. 5 provides the workflow and an example of synthetic environment generation using LLM. In this example, the LLM generates a variant of the original simulated environment based on the official vulnerability description and original simulation $\widetilde{\mathcal{M}}_0$. In this variant, the web application (Drupal) is exposed on a non-default port (9000), and there are random changes in the operating system version, Apache HTTP Server version, and Drupal version compared to the original simulation. These changes achieve domain randomization, making the synthetic environment better replicate the diversity found in the real world, thereby preventing agents from relying on fixed host configuration details.

You are a penetration testing expert with information on various vulnerabilities. Now we are trying to scan a target host with the **{CVE_ID}** vulnerability using tools such as Nmap and WhatWeb. The **{CVE_ID}** vulnerability is described as follows: **{vul_description}**. The following is an example of the scan result: **{example}**. Since the target hosts with **{CVE_ID}** vulnerability may be scanned differently due to different host configurations (like port, services, OS, or web_fingerprint), please follow the given example, just generate five different scan results in JSON format, and explain why these results are reasonable.

Fig. 4 Prompt for synthetic environment generation

4.4 Meta-RL training

Given the set of synthetic environments $\widetilde{\mathcal{M}} = \{\widetilde{\mathcal{M}}_1, \widetilde{\mathcal{M}}_2, ...\}$ generated by the LLM on the basis of $\widetilde{\mathcal{M}}_0$, in this part, we leverage $\widetilde{\mathcal{M}}$ as meta-training environments and employ the model-agnostic meta-learning (MAML) algorithm (Finn et al., 2017) for

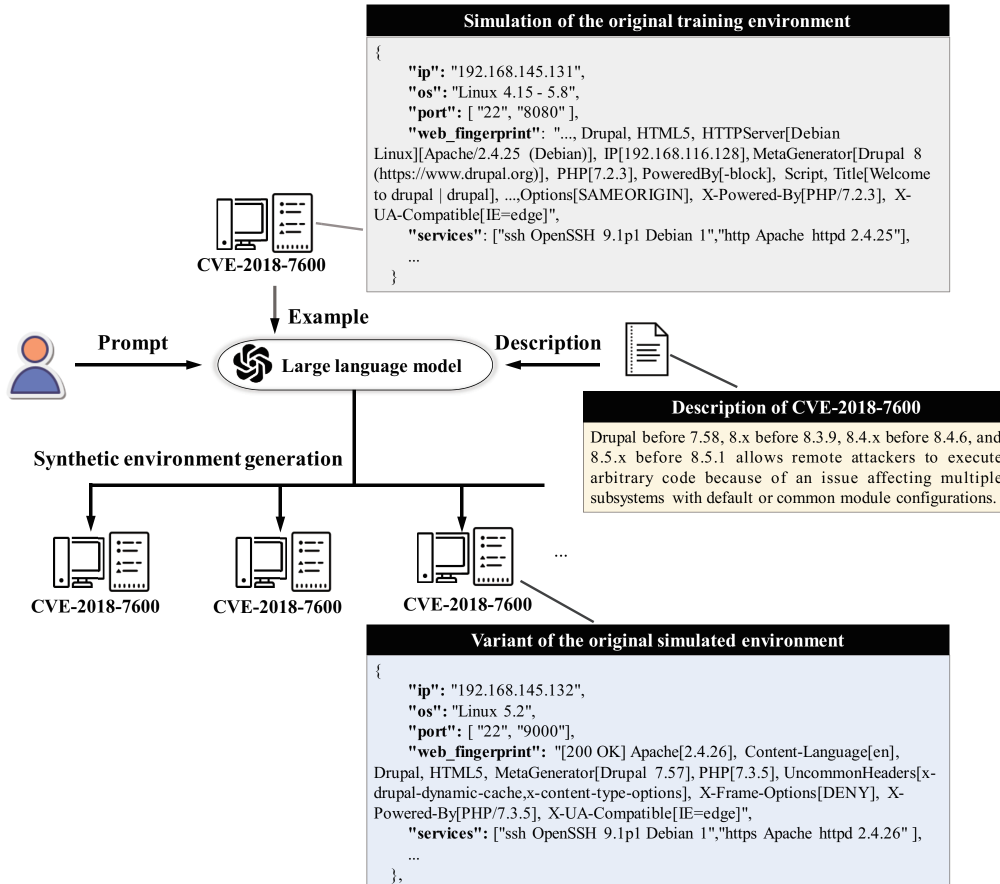

Fig. 5 The workflow and example of synthetic environment generation using a large language model. In this example, we construct the simulation by using the CVE-2018-7600 vulnerable host from Vulhub as the original training environment. Then, we use GLM-4 (Team GLM, 2024) for domain randomization

meta-RL training. It is important to note that while MAML is used here, a wide range of other meta-RL algorithms are also applicable.

In MAML, the inner loop is a policy gradient algorithm with initial parameters as meta-parameters $\phi = \theta$. The key insight of MAML is that its inner loop is a differentiable function of the initial parameters (Beck et al., 2023). This means that the initialization of the model can be optimized using gradient descent to find a set of initial parameters that serve as a good starting point for learning new tasks drawn from the task distribution. During the new task adaptation phase, MAML employs an on-policy policy gradient algorithm to update the policy parameters. This requires MAML to sample new trajectories $\mathcal{D}_i$ using the initial policy, and then use these trajectories to update a set of parameters by applying a policy gradient step for a task/environment $\widetilde{\mathcal{M}}_i \in \widetilde{\mathcal{M}}$:

$$\phi_i' = \phi + \alpha \bigtriangledown_\phi \hat{J}_{\widetilde{\mathcal{M}}_i}(\pi_\phi, \mathcal{D}_i), \tag{6}$$

where $\hat{J}_{\widetilde{\mathcal{M}}_i}(\pi_\phi, \mathcal{D}_i)$ is an estimate of the expected discounted reward of policy π_ϕ for task $\widetilde{\mathcal{M}}_i$, and α is the learning rate. After adapting to each task, MAML collects new trajectories $\mathcal{D}_i'$ again using adapted policy $\pi_{\phi_i'}$ for updating the initial parameters ϕ in the outer loop:

$$\phi \leftarrow \phi + \beta \bigtriangledown_\phi \sum_{\widetilde{\mathcal{M}}_i \in \widetilde{\mathcal{M}}} \hat{J}_{\widetilde{\mathcal{M}}_i}(\pi_{\phi_i'}, \mathcal{D}_i'), \tag{7}$$

where $\pi_{\phi_i'}$ is the policy for task $\widetilde{\mathcal{M}}_i$ adapted by the inner loop, and $\bigtriangledown_\phi \hat{J}_{\widetilde{\mathcal{M}}_i}(\pi_{\phi_i'}, \mathcal{D}_i')$ is the gradient of the expected discounted reward of the adapted policy calculated with respect to the initial parameters. The gradient collected from the inner loop is referred to as the meta-gradient, which reflects how the agent's performance on the new task affects the updates to the agent's policy parameters. Descending the meta-gradient is essentially performing gradient descent on the meta-RL objective given by Eq. (2). The complete meta-RL training algorithm is presented in Algorithm 1.

Algorithm 1 Meta-RL training with MAML

Require: Meta-training environments $\widetilde{\mathcal{M}}$
Require: Learning rates α and β, and the initial policy parameters ϕ
1: **for** each task environment $\widetilde{\mathcal{M}}_i \in \widetilde{\mathcal{M}}$ **do**
2: Sample trajectories $\mathcal{D}_i$ using policy π_ϕ in $\widetilde{\mathcal{M}}_i$
3: Compute the adapted parameters with gradient descent following Eq. (6)
4: Sample new trajectories $\mathcal{D}_i'$ using policy $\pi_{\phi_i'}$ in $\widetilde{\mathcal{M}}_i$
5: **end for**
6: Update policy parameters ϕ following Eq. (7) using each $\mathcal{D}_i'$

5 Simulation settings

5.1 Pentesting environments

To avoid potentially unpredictable impacts on real-world hosts and better evaluate the performance of GAP in real scenarios, we conduct simulations in high-fidelity vulnerability environments set up using Vulhub. Vulhub is an open-source collection of vulnerable Docker environments that offers a flexible way to easily create various pentesting environments. In this way, we can balance simulation control with the validation of real-world performance.

Table 1 lists all simulation vulnerability environments sourced from Vulhub, each serving as an original training environment $\mathcal{M}_0^k$ in GAP, where k represents the CVE identifier. We select these vulnerability environments for training and testing the performance of the pentesting agent because they target various representative VPs. More importantly, we have reliable and stable exploits of these vulnerabilities available for the agent to use. Theoretically, GAP can be extended to a broader range of vulnerability environments.

5.2 Agent settings

In each environment, the agent is trained for 500 episodes with a maximum of 100 iteration steps per episode. During the pentesting process, the agent starts in an initial state without prior knowledge of the target host. Its task is to gather information and then exploit the corresponding vulnerability to compromise the host within limited steps. Its available actions include various information scanning tools (e.g., Nmap, WhatWeb, and Dirb) and exploits in Metasploit Framework (MSF). In this paper, we change the size of the action space $|\mathcal{A}|$ by randomly sampling different numbers of vulnerability exploits from MSF.

Following Zhou et al. (2024), our reward function is formulated based on subjective estimates provided by human experts. Specifically, a reward of +1000 is assigned when the agent successfully exploits the correct vulnerability and compromises the

Table 1 Vulnerability environments

CVE	Vulnerability name	Vulnerable product
CVE-2018-7600	Drupal Core Remote Code Execution Vulnerability	Drupal Core
CVE-2019-0230	Apache Struts OGNL Remote Code Execution Vulnerability	Apache Struts
CVE-2019-9193	PostgreSQL Authenticated Remote Code Execution Vulnerability	PostgreSQL
CVE-2020-10199	Sonatype Nexus Repository Remote Code Execution Vulnerability	Nexus
CVE-2020-7247	OpenSMTPD Remote Code Execution Vulnerability	OpenSMTPD
CVE-2020-7961	Liferay Portal Deserialization of Untrusted Data Vulnerability	Liferay Portal
CVE-2020-9496	Apache OFBiz Remote Code Execution Vulnerability	Apache OFBiz
CVE-2020-16846	SaltStack Salt Shell Injection Vulnerability	SaltStack Salt
CVE-2021-22205	GitLab Unauthenticated Remote Code Execution Vulnerability	GitLab
CVE-2021-25646	Apache Druid Remote Code Execution Vulnerability	Apache Druid
CVE-2021-3129	Laravel Ignition File Upload Vulnerability	Ignition
CVE-2021-41773	Apache HTTP Server Path Traversal Vulnerability	Apache HTTP Server
CVE-2022-0543	Debian-specific Redis Server Lua Sandbox Escape Vulnerability	Redis
CVE-2022-22947	VMware Spring Cloud Gateway Code Injection Vulnerability	Spring Cloud Gateway
CVE-2022-46169	Cacti Command Injection Vulnerability	Cacti
CVE-2023-21839	Oracle WebLogic Server Unspecified Vulnerability	Oracle WebLogic
CVE-2023-32315	Ignite Realtime Openfire Path Traversal Vulnerability	Openfire
CVE-2023-38646	Metabase Pre-Auth Remote Code Execution Vulnerability	Metabase
CVE-2023-42793	JetBrains TeamCity Authentication Bypass Vulnerability	JetBrains
CVE-2023-46604	Apache ActiveMQ Deserialization of Untrusted Data Vulnerability	Apache ActiveMQ

target host. Additionally, a reward of +100 is allocated for proficiently gathering information about the target, such as the OS type, version, running services, and website fingerprint. Conversely, incorrect or illogical actions result in a penalty of −10.

5.3 Implementation details

All vulnerability environments are deployed on virtual machines. We train the pentesting agent on a 64-bit laptop powered by an Intel® Core™ i9-12900H CPU @ 2.50 GHz, with 32 GB of memory, and an NVIDIA GeForce RTX 3060 laptop GPU. The laptop runs Kali Linux and is equipped with MSF version v6.4.5-dev. PyTorch is used as the backend for implementing GAP.

In GAP, we use the classical PPO algorithm for policy learning and the MAML algorithm for meta-RL training. The synthetic environments used for meta-RL training are generated using GLM-4 (Team GLM, 2024) due to its satisfactory performance, and we assess their validity based on expert evaluators who are familiar with real-world pentesting scenarios. In fact, as a simple and general framework, GAP is not restricted to specific RL or meta-RL algorithms, nor is it limited to particular LLMs.

Our code, training and testing environments, and hyperparameters are publicly available for further replicability and future research, and are available at https://github.com/Joe-zsc/GAP.

5.4 Research questions

GAP aims to improve the agent's generalization ability, thereby allowing the agent to draw inferences about other cases from one instance. We aim to answer the following research questions (RQs) from simulations and analysis:

RQ1 (learn from one): can GAP perform policy learning in various real environments?

RQ2 (generalize to another): can GAP bridge the generalization gap and achieve zero-shot policy transfer in similar environments?

RQ3 (generalize to another): can GAP achieve rapid policy adaptation in dissimilar environments?

5.5 Evaluation metrics

5.5.1 Evaluation of RQ1

To investigate RQ1, in each run, we evaluate the agent's learning curve and training time for all the original training environments as listed in Table 1. We compare the performance of the agent with different sizes of action spaces ($|\mathcal{A}| \in \{100, 500, 1000\}$). We implement three independent runs using different seeds for each vulnerability environment, with the average results shown in Section 6.1.

5.5.2 Evaluation of RQ2

To investigate RQ2, following Lyle et al. (2022) and Kirk et al. (2023), we assess the agent's zero-shot generalization performance in both training and testing environments, measuring its ability to bridge the generalization gap (GenGap, calculated by Eq. (3)). In addition, the average success rate is used as a metric, which reflects the proportion of hosts successfully compromised by the agent relative to the total number of hosts in the testing environments.

To this end, we create three variants for each $\mathcal{M}_0^k$ in Table 1 by manually changing its host configurations to serve as testing environments denoted by $\mathcal{M}_{\text{test}}^k = \left\{\mathcal{M}_1^k, \mathcal{M}_2^k, \mathcal{M}_3^k\right\}$. These variants in $\mathcal{M}_{\text{test}}^k$ are similar to $\mathcal{M}_0^k$ as they share the same vulnerability k, but they differ in host configurations such as ports, services, OS versions, and website fingerprints. These differences create reality gaps that mimic the diversity and variability found in real-world host environments.

In each implementation run, we train the agent for each $\mathcal{M}_0^k$ in Table 1, and later evaluate its zero-shot performance in $\mathcal{M}_0^k$ and testing environments $\mathcal{M}_{\text{test}}^k$, as well as their pentesting success rate in $\mathcal{M}_{\text{test}}^k$. We implement three independent runs using different seeds for each vulnerability environment, with the average results shown in Section 6.2.

5.5.3 Evaluation of RQ3

A well-generalized agent should also be able to effectively improve its learned policy to adapt to a new environment. To investigate RQ3, following Kirk et al. (2023), we evaluate the agent's few-shot policy adaptation performance in dissimilar environments by allowing the agent to interact with a new environment online.

In particular, in each run, for each $\mathcal{M}_0^k$ in Table 1, we randomly select another environment $\mathcal{M}_0^{k'}$ as the testing environment, where $k' \neq k$. The agent is trained in $\mathcal{M}_0^k$ and transfers its learned policy in $\mathcal{M}_0^{k'}$. We evaluate its learning curve and training time in the testing environment, which measures how quickly the agent can adapt its performance to a new task.

In addition, following Zhu et al. (2023), we evaluate the agent's jumpstart performance, which refers to the agent's initial performance when it starts interacting with the testing environment. It assesses how quickly the agent can abstract broader knowledge from past experiences and apply it to novel situations, thereby adapting its policy and achieving meaningful rewards in the early stages of training. We implement three independent runs using different seeds for each vulnerability environment, with the average results shown in Section 6.3.

6 Simulation results

6.1 Policy learning in the real environments (RQ1)

Being able to learn in various real environments is the foundation skill for pentesting agents. In this part, we train the agent to learn policies from scratch in all original training environments that are unknown to the agent. We compare the learning performance of the agent across different action spaces ($|\mathcal{A}| \in \{100, 500, 1000\}$). Figs. 6 and 7 display the agent's learning curve and training time, respectively.

The simulation results demonstrate that the agent can learn policies in training environments. However, as the action space increases, there is a decrease in the learning efficiency, a slower convergence rate of the learning curve, and an increase in the training time accordingly. This phenomenon arises because a larger action space requires more extensive exploration to identify effective actions, thereby prolonging the time needed for the agent to discover the optimal policy. Additionally, optimizing the policy model becomes more computationally demanding with increased iterations and computational effort as the action space expands.

As more vulnerabilities are disclosed, the agent inevitably requires a larger action space to address the expanding attack surface. Consequently, enhancing the learning efficiency of the agent in larger action spaces becomes an interesting research topic. This paper proposes a solution by enhancing the agent's generalization ability. This enables the agent to draw inferences across environments, achieving zero-shot generalization in similar settings and rapid policy adaptation in dissimilar ones. The simulation results supporting these claims are presented in the following sections.

During interaction with the original training environments, the agent can gather the host

configuration data (e.g., ports, services, and website fingerprint) and save them in JSON format as the simulated copy of the real environment. This approach not only provides realistic simulated environments for agent training, effectively addressing the training environment dilemma, but also enables the synthesis of additional diverse simulation environments using LLMs, thereby augmenting the environmental variety. An example of this process is shown in Fig. 5, and more details of the simulated environments are available in our code repository.

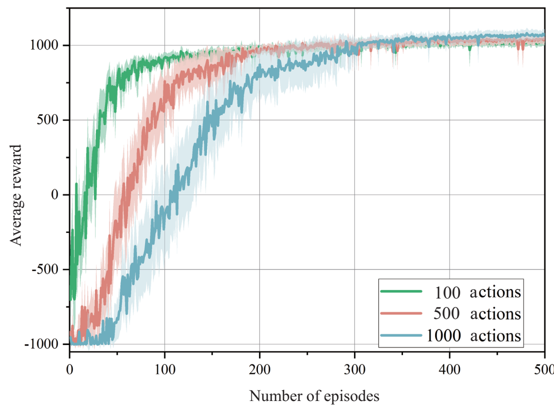

Fig. 6 Learning curves of the agent with varying action space sizes in all original training environments. Each curve is the average reward over three independent runs, while the shaded area denotes the 95% confidence intervals. References to color refer to the online version of this figure

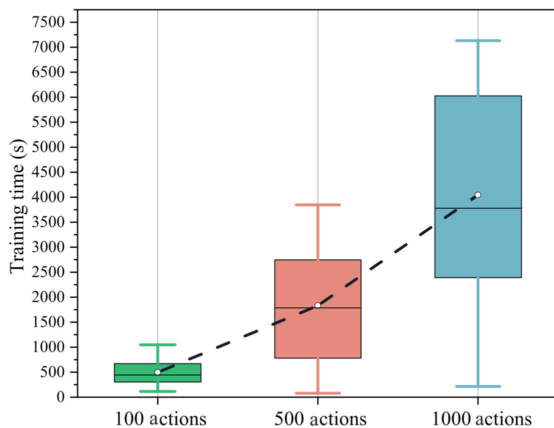

Fig. 7 Training time of the agent with varying action space sizes in all original training environments. The box plot summarizes the distribution of training time across 500 episodes for each action space. The dashed line connects the average training time. References to color refer to the online version of this figure

6.2 Zero-shot policy transfer in similar environments (RQ2)

In this part, we demonstrate the agent's zero-shot generalization ability in testing environments that have the same vulnerability as the original training environment.

In particular, the agent is pre-trained using various methods in the original training environments and subsequently transfers its policies to its training environment and testing environments. We compare the zero-shot generalization performance between GAP and two baseline methods: PPO (Yang et al., 2025) and APRIL (Zhou et al., 2024). Both baselines are state-of-the-art methods that have been widely used in autonomous pentesting.

In addition, we analyze the effect of the number of meta-training environments on the zero-shot generalization performance of GAP. We use n-GAP to denote the use of n meta-training environments in the GAP, where $n \in \{3, 5, 8\}$ in our simulations.

As shown in Fig. 8, when the agent's policy is transferred to the original training environments, all methods maintain a satisfactory performance. However, when the policy is transferred to the testing environments, both baselines experience a noticeable performance loss. This performance gap between the training and testing environments is referred to as the generalization gap. Table 2 presents the generalization gap (GenGap) of different methods, along with their success rate in the testing environments. From the results, we can see that our proposed framework, GAP, significantly enhances the success rate in testing environments compared to the baseline methods and bridges the generalization gap, demonstrating improved zero-shot generalization ability.

These results indicate that the use of domain randomization increases both the quantity and the diversity of training environments. This exposure enables the agent to adapt to a wider range of environmental variations during training. Through

Table 2 Generalization gap and success rate in the testing environments

Method	GenGap	Success rate
APRIL	990.21	0.14
PPO	624.33	0.44
3-GAP	171.68	0.83
5-GAP	51.61	0.92
8-GAP	61.11	0.91

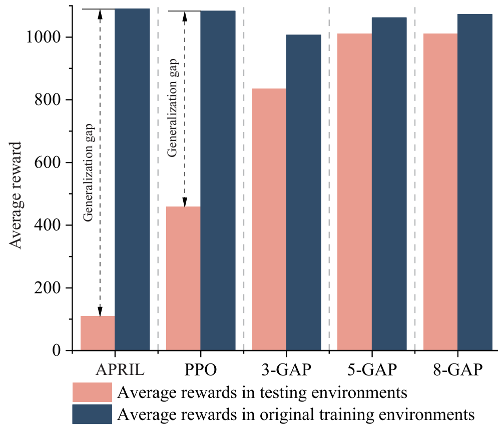

Fig. 8 Zero-shot generalization performance in the original training environments and testing environments

meta-RL, the agent learns how to extract generalizable policies and biases from the diverse meta-training environments, thereby allowing the agent to effectively generalize its learned policies to new, similar testing environments.

Additionally, fewer meta-training environments affect the agent's zero-shot generalization ability to some extent. However, increasing the number of meta-training environments up to a certain point does not significantly improve the generalization performance. This could be due to the fact that the agent reaches a saturation point where additional environment variations do not significantly contribute to further learning. Specifically, our simulations reveal that the agent can achieve satisfactory zero-shot generalization performance when trained with five meta-training simulations, which serves as the default setting in the subsequent simulations.

6.3 Few-shot policy adaptation in dissimilar environments (RQ3)

In this part, we evaluate the agent's policy adaptation ability when faced with testing environments that are dissimilar to the original training environments. The policy adaptation ability refers to the agent's ability to quickly adapt its learned policy to unseen environments through policy transfer.

We use the following four methods to train the agent with action space $|A| = 1000$:

1. Learn from scratch. Similar to Section 6.1, the agent is directly trained using PPO in testing environments without policy transfer.

2. PPO-Transfer. We pre-train the agent using PPO in the original training environments and then transfer its learned policy to testing environments.

3. APRIL-Transfer. We pre-train the agent using APRIL in the original training environments and then transfer its learned policy to testing environments. APRIL is a state-of-the-art autonomous pentesting framework that has been proven to exhibit strong transfer learning performance (Zhou et al., 2024).

4. GAP-Transfer. Following the real-to-sim-to-real pipeline, the agent is pre-trained in the original training environments, and then its generalization is improved by adopting domain randomization and meta-RL training. Finally, we transfer the learned policy to testing environments.

Figs. 9 and 10 display the agent's learning curve and training time, respectively, in testing environments. The results show that compared to learning from scratch, the other three methods enhance the convergence speed and reduce the training time through policy transfer. Among them, GAP stands out as the most effective one, reducing the average training time by approximately 40% (from 4045 to 2429 s).

Furthermore, as shown in Figs. 9 and 10, it can be observed that while GAP-Transfer and APRIL-Transfer exhibit similar convergence speeds, GAP-Transfer reduces the training time by approximately 22% compared to APRIL-Transfer (from 3129 to 2429 s). This reduction is related to the jumpstart performance, as evidenced by Table 3 and Fig. 11.

Table 3 presents the jumpstart performance of different methods, from which we can see that GAP-Transfer demonstrates superior jumpstart

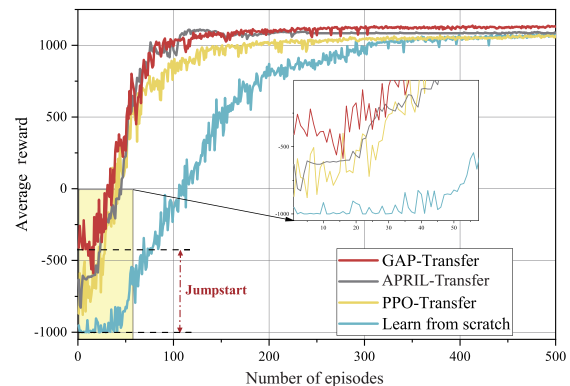

Fig. 9 Learning curves of different methods in testing environments. References to color refer to the online version of this figure

performance than others. In addition, Fig. 11 illustrates a strong negative linear correlation between jumpstart performance and average training time, with a Pearson correlation coefficient (Pearson's r) close to -0.99. A better jumpstart allows the agent to explore effective actions more quickly during the initial training in new environments, indicating that the agent can generalize knowledge from past experiences and apply it effectively to new situations. The outstanding performance of GAP-Transfer can be attributed to the incorporation of a meta-learning mechanism, which empowers the agent to improve its policy adaptation ability by learning how to learn.

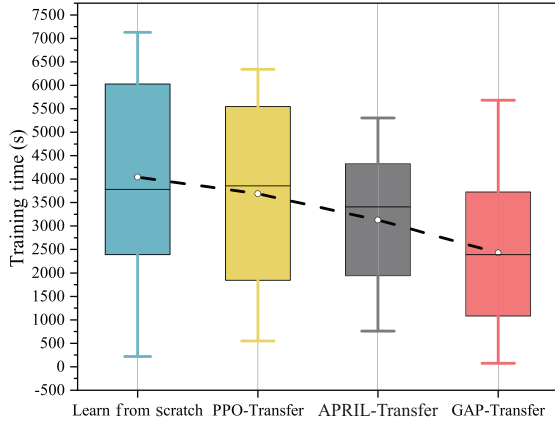

Fig. 10 Training time of the agent in testing environments. The box plot summarizes the distribution of training time across 500 episodes, while the dashed line connects the average value. References to color refer to the online version of this figure

Table 3 Jumpstart performance of different methods

Method	Jumpstart performance
GAP-Transfer	589.22
APRIL-Transfer	367.21
PPO-Transfer	129.83
Learn from scratch	47.33

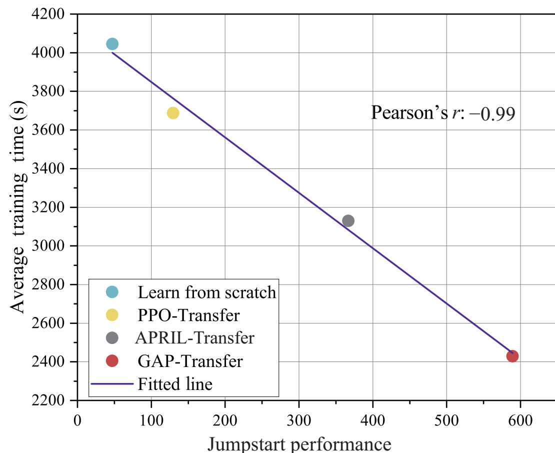

Fig. 11 Correlation between the jumpstart performance and average training time of different methods. References to color refer to the online version of this figure

7 Discussion

First, our work aims to enhance the policy generalization ability of pentesting agents, which is essential for their widespread applicability. This paper primarily explores the use of meta-RL, combined with domain randomization techniques, as a direct solution to enhance agents' generalization ability. In the future, additional methods, such as multi-task learning (Huang et al., 2023) or domain transfer (Zhu et al., 2023) techniques, can be employed to further improve generalization capabilities. Moreover, another intriguing area that demands exploration is achieving "lifelong learning" in agents. Lifelong learning (Parisi et al., 2019) focuses on enabling agents to avoid forgetting past tasks when faced with numerous new tasks.

Additionally, this paper provides a feasible solution for training pentesting agents on actual vulnerable machines. Given the constraints of real-world environment safety and the availability of exploit code, we validate the application potential of these agents in real environments by testing them in a limited number of virtual machine environments. These environments predominantly feature vulnerabilities that enable remote command execution. Admittedly, our vulnerability environments are somewhat idealized and overlook the impacts that complex real-world environments may have on pentesting agents, such as effects from network protocols or honeypot environments. Effective autonomous pentesting by agents in complex environments hinges on more precise environmental observation capabilities and a rich, fine-grained action space. This will be a focal point of our future research endeavors.

Furthermore, in GAP, we leverage LLMs for domain randomization to enhance the environmental diversity. LLMs are prone to generating hallucinations that could result in unrealistic or logically inconsistent simulated scenarios. This, in turn, may negatively affect the quality of the agent's learning process. To mitigate this issue, we rely on human

experts to evaluate the quality of the simulated environments generated by the LLMs. Expert evaluations help us identify and exclude scenarios that contain obvious errors or logical inconsistencies, reducing the risk of training the agent in flawed environments. In the future, one promising direction is the use of retrieval-augmented generation (RAG) techniques (Shuster et al., 2021), which could help reduce hallucinations by grounding the model's generation process with more reliable external data. We plan to explore this approach in future work to further enhance the quality of the simulated environments.

Finally, in our work, the reward function's design relies on experts' subjective estimates. However, assessing the reward function presents a challenge since there is no objective standard defining an "optimal" strategy (Holm, 2023). One potential avenue for future research is to explore automated methods (e.g., inverse reinforcement learning (Metelli, 2024)) for generating reward functions.

8 Conclusions

In this paper, we present GAP, an autonomous pentesting framework for efficient policy training in realistic environments and for training generalizable agents capable of drawing inferences about other cases from one instance—a key to the broad application of autonomous pentesting agents. To achieve this, GAP introduces a real-to-sim-to-real pipeline that (1) enables end-to-end policy learning in unknown realistic environments while constructing realistic simulation analogs and (2) improves agents' generalization ability by leveraging domain randomization and meta-RL learning. The preliminary evaluations demonstrate that GAP allows pentesting agents for end-to-end policy learning in realistic environments, bridging the generalization gap for zero-shot policy transfer in similar environments, and facilitating rapid policy adaptation in dissimilar environments.

Contributors

Shicheng ZHOU and Yue ZHANG designed the research. Yue ZHANG processed the data. Shicheng ZHOU and Yue ZHANG drafted the paper. Jingju LIU, Jie CHEN, and Yuliang LU helped organize the paper. Jingju LIU, Yuliang LU, and Jiahai YANG revised and finalized the paper.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

Our code, training and testing environments, and hyperparameters are publicly available for further replicability and future research, and are available at https://github.com/Joe-zsc/GAP.

References

Beck J, Vuorio R, Liu EZ, et al., 2023. A survey of meta-reinforcement learning. https://doi.org/10.48550/ARXIV.2301.08028

Bo L, Zhang TZ, Zhang HX, et al., 2024. 3D UAV path planning in unknown environment: a transfer reinforcement learning method based on low-rank adaption. *Adv Eng Inform*, 62:102920. https://doi.org/10.1016/J.AEI.2024.102920

Chen J, Wu DD, Xie RY, 2023. Artificial intelligence algorithms for cyberspace security applications: a technological and status review. *Front Inform Technol Electron Eng*, 24(8):1117-1142. https://doi.org/10.1631/FITEE.2200314

Chen JY, Hu SL, Zheng HB, et al., 2023. GAIL-PT: an intelligent penetration testing framework with generative adversarial imitation learning. *Comput Secur*, 126:103055. https://doi.org/10.1016/j.cose.2022.103055

Chen XY, Hu JC, Jin C, et al., 2022. Understanding domain randomization for sim-to-real transfer. Proc 10th Int Conf on Learning Representations, p.1-28.

Cobbe K, Klimov O, Hesse C, et al., 2019. Quantifying generalization in reinforcement learning. https://arxiv.org/abs/1812.02341

Feng S, Sun HW, Yan XT, et al., 2023. Dense reinforcement learning for safety validation of autonomous vehicles. *Nature*, 615:620-627. https://doi.org/10.1038/s41586-023-05732-2

Finn C, Abbeel P, Levine S, 2017. Model-agnostic meta-learning for fast adaptation of deep networks. Proc 34th Int Conf on Machine Learning, p.1126-1135.

Guo X, Chen YQ, 2024. Generative AI for synthetic data generation: methods, challenges and the future. https://doi.org/10.48550/ARXIV.2403.04190

Holm H, 2023. Lore a red team emulation tool. *IEEE Trans Depend Secur Comput*, 20(2):1596-1608. https://doi.org/10.1109/TDSC.2022.3160792

Horváth D, Erdös G, Istenes Z, et al., 2023. Object detection using sim2real domain randomization for robotic applications. *IEEE Trans Robotics*, 39(2):1225-1243. https://doi.org/10.1109/TRO.2022.3207619

Hospedales TM, Antoniou A, Micaelli P, et al., 2022. Meta-learning in neural networks: a survey. *IEEE Trans Pattern Anal Mach Intell*, 44(9):5149-5169. https://doi.org/10.1109/TPAMI.2021.3079209

Huang HC, Ye DH, Shen L, et al., 2023. Curriculum-based asymmetric multi-task reinforcement learning. *IEEE Trans Pattern Anal Mach Intell*, 45(6):7258-7269. https://doi.org/10.1109/TPAMI.2022.3223872

Ilic N, Dasic D, Vucetic M, et al., 2024. Distributed web hacking by adaptive consensus-based reinforcement learning. *Artif Intell*, 326:104032.
https://doi.org/10.1016/J.ARTINT.2023.104032

Jonathon S, Hanna K, 2019. NetworkAttactSimulator. https://github.com/Jjschwartz/NetworksttackSimulator [Accessed on Feb. 16, 2025].

Kirk R, Zhang A, Grefenstette E, et al., 2023. A survey of zero-shot generalisation in deep reinforcement learning. *J Artif Intell Res*, 76:201-264.
https://doi.org/10.1613/JAIR.1.14174

Li QY, Wang RP, Li D, et al., 2024. DynPen: automated penetration testing in dynamic network scenarios using deep reinforcement learning. *IEEE Trans Inform Forens Secur*, 19:8966-8981.
https://doi.org/10.1109/TIFS.2024.3461950

Li ZY, Zhu HX, Lu ZR, et al., 2023. Synthetic data generation with large language models for text classification: potential and limitations. Proc Conf on Empirical Methods in Natural Language Processing, p.10443-10461.
https://doi.org/10.18653/V1/2023.EMNLP-MAIN.647

Lyle C, Rowland M, Dabney W, et al., 2022. Learning dynamics and generalization in deep reinforcement learning. Proc Int Conf on Machine Learning, p.14560-14581.

Maeda R, Mimura M, 2021. Automating post-exploitation with deep reinforcement learning. *Comput Secur*, 100:102108.

Metelli AM, 2024. Recent advancements in inverse reinforcement learning. Proc 38th AAAI Conf on Artificial Intelligence, p.22680.
https://doi.org/10.1609/AAAI.V38I20.30296

Microsoft Defender Research Team, 2021. CyberBattleSim. https://github.com/microsoft/cyberbattlesim [Accessed on Feb. 16, 2025].

Nguyen HPT, Hasegawa K, Fukushima K, et al., 2025. PenGym: realistic training environment for reinforcement learning pentesting agents. *Comput Secur*, 148:104140.
https://doi.org/10.1016/J.COSE.2024.104140

Parisi GI, Kemker R, Part JL, et al., 2019. Continual lifelong learning with neural networks: a review. *Neur Netw*, 113:54-71.
https://doi.org/10.1016/J.NEUNET.2019.01.012

Schulman J, Wolski F, Dhariwal P, et al., 2017. Proximal policy optimization algorithms.
https://doi.org/10.48550/arXiv.1707.06347

Shuster K, Poff S, Chen MY, et al., 2021. Retrieval augmentation reduces hallucination in conversation. Proc Findings of the Association for Computational Linguistics, p.3784-3803.
https://doi.org/10.18653/V1/2021.FINDINGS-EMNLP.320

Takaesu I, 2018. DeepExploit. https://github.com/13o-bbr-bbq/machine_learning_security/blob/master/DeepExploit [Accessed on Feb. 16, 2025].

Team GLM, 2024. ChatGLM: a family of large language models from GLM-130B to GLM-4 All Tools.
https://doi.org/10.48550/arXiv.2406.12793

Tobin J, Fong R, Ray A, et al., 2017. Domain randomization for transferring deep neural networks from simulation to the real world. Proc IEEE/RSJ Int Conf on Intelligent Robots and Systems, p.23-30.
https://doi.org/10.1109/IROS.2017.8202133

Tran K, Akella A, Standen M, et al., 2021. Deep hierarchical reinforcement agents for automated penetration testing.
https://doi.org/10.48550/arXiv.2109.06449

Wang KX, Kang BY, Shao J, et al., 2020. Improving generalization in reinforcement learning with mixture regularization. Proc Annual Conf on Neural Information Processing Systems, p.7968-7978.

Wang KX, Reimers N, Gurevych I, 2021. TSDAE: using Transformer-based sequential denoising auto-encoderfor unsupervised sentence embedding learning. Proc Findings of the Association for Computational Linguistics, p.671-688.
https://doi.org/10.18653/V1/2021.findings-emnlp.59

Yang YZ, Chen MX, Fu HH, et al., 2023. SetTron: towards better generalisation in penetration testing with reinforcement learning. Proc IEEE Global Communications Conf, p.4662-4667.
https://doi.org/10.1109/globecom54140.2023.10437804

Yang YZ, Chen LD, Liu S, et al., 2025. Behaviour-diverse automatic penetration testing: a coverage-based deep reinforcement learning approach. *Front Comput Sci*, 19(3):193309.
https://doi.org/10.1007/s11704-024-3380-1

Ye DY, Zhu TQ, Gao K, et al., 2024. Defending against label-only attacks via meta-reinforcement learning. *IEEE Trans Inform Forens Secur*, 19:3295-3308.
https://doi.org/10.1109/TIFS.2024.3357292

Zhao WS, Queralta JP, Westerlund T, 2020. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. Proc IEEE Symp Series on Computational Intelligence, p.737-744.
https://doi.org/10.1109/SSCI47803.2020.9308468

Zhou SC, Liu JJ, Lu YL, et al., 2024. APRIL: towards scalable and transferable autonomous penetration testing in large action space via action embedding. *IEEE Trans Depend Secur Comput*, 22(3):2443-2459.
https://doi.org/10.1109/TDSC.2024.3518500

Zhu ZD, Lin KX, Jain AK, et al., 2023. Transfer learning in deep reinforcement learning: a survey. *IEEE Trans Pattern Anal Mach Intell*, 45(11):13344-13362.
https://doi.org/10.1109/TPAMI.2023.3292075